Haaga-Helia
ammattikorkeakoulu Oy

Ketteryyttä kehittämiseen, case: automatisoitava elinkaa-ritestaus

Tuomas Törmä

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma
2016

Haaga-Helia ammattikorkeakoulu Oy

Tiivistelmä

17.5.2016

Tekijä(t)
Tuomas Törmä

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön otsikko
Ketteryyttä kehittämiseen, case: automatisoitava elinkaaritestaus

Sivu- ja liitesivumäärä
42 + 6

Opinnäytetyön otsikko englanniksi
Agile developing, case: automated software lifecycle testing

Opinnäytetyö oli Sp-Henkivakuutus Oy:n tilaama toimeksianto, jossa tavoitteena oli luoda automatisoitava ratkaisu sisäisten ohjelmiston elinkaaren testaamiseen, testauskulujen keskittämiseen ja laadun mittaamiseen. Tulostavoitteena oli rakentaa ohjelma, jonka avulla voitaisiin testata ohjelmiston elinkaarta automaation kautta luoden tarvittavat syötteet testattavalle tietojärjestelmälle. Luodut syötetehtävät validoitiin ja verifioitiin niiden omien vaatimusmääritysten mukaisesti, jotta testattavalla ohjelmalla olisi ollut mahdollisuus ajaa syötteet tietojärjestelmään. Lisäksi ohjelman luomat tehtävät tulee pystyä auditoida myöhempää tutkimusta varten, mikäli on tarvetta.

Teoria opinnäytetyöhön kerättiin sekä Internet-lähteistä että kirjallisista teoksista. Tätä tietoa käytettiin hyväksi ketterässä ohjelmistokehityksessä ja luotaessa elinkaarta testaavaa ohjelmistoa.

Ohjelmiston kehitys tehtiin ketteränä ohjelmistokehitysprojektina, jossa oli myös Lean-ajattelun ja DevOps-prosessin piirteitä. Ohjelmiston laadun ja jatkuvan parantamisen kannalta käytettiin jatkuvaa integrointia sekä öisin SonarQube analysoi lähdekoodin mahdollisten kehitysongelmien ja vikojen löytämiseksi. Projektia hallinnointiin kevennetyn Scrum-ideologian puitteissa. Koko projekti koostui neljästä iterointikierroksesta alkaen suunnitteluvaiheella ja jokaisen iterointikierroksen lopussa näyttötilaisuus pidettiin asiakkaan toimitiloissa. Näyttötilaisuuksien jälkeen pidettiin retrospektiivit ja seuraava iterointikierros suunniteltiin.

Opinnäytetyössä käytiin läpi suunnilleen kaikki ketterän kehittämisen näkökulmat ruohonjuuritasolta aina ohjelmiston julkaisuun. Kehitysprosessia valvottiin toimeksiantajan omilla laitteilla. Loppujen lopuksi ohjelmistolla ajettiin ensimmäiset testikierrokset onnistuneesti.

Asiasanat
Ohjelmiston elinkaari, testaus, sovelluskehys, Java, laatu, automaatio

Haaga-Helia
University of Applied Sciences

Abstract

17.5.2016

Author(s)
Tuomas Törmä

Degree programme
Business information Technology

Report/thesis title
Agile developing, case: automated software lifecycle testing

Number of pages and appendix pages
42 + 6

This bachelor's thesis was commissioned by the company called Sb Life Insurance Ltd. The aim of this project was to develop automation software that could be used to test the company's own information software in order to focus the test process and create quality meters. The Assignment included creation of the automation software that can be used for testing another software's life cycle by creating input files for the system. These created input tasks had to also be validated and verified by their requirement specification so that they can be executed by the tested software. In addition, these individually executed tasks, which were built by the testing system, should be audited, so they could be analysed for further studies if need to be.

The research material for this thesis was gathered from various online and literary sources. The theoretical background knowledge was applied in agile software development and also in creating the testing life cycle system.

The software development was carried out as an agile software development project inspired by Lean thinking and DevOps process. In addition, continuous integration was used to constantly improve the software quality aspects and the source code was regularly analysed by SonarQube to find out pitfalls and defects in the development process. The test coverage was examined by the SonaQube. In order to manage the project, lightened Scrum was used. The project consisted of four iterations starting with the planning phase and for each iteration an end phase review was held on the commissioner's premises. After reviews, a retrospective was held by the developing team and the next iteration was planned.

In conclusion, the thesis covered mostly all aspects of agile software development from grass roots to releasing software. The development process was constantly monitored and viewed by the commissioner own systems. In the end, the software was used to create first test phase round to meet the commissioner's expectations.

Keywords
Software lifecycle, testing, framework, Java, quality, automation

Käytetty termistö

Automatisoitava testaus

Testaus, joka voidaan halutessa automatisoida. Testitapaukset ovat luonteeltaan sellaisia, että ne voidaan automatisoida. Testaus voidaan suorittaa myös manuaalisesti.

Elinkaari

Elinkaarella tarkoitetaan tässä työssä vakuutussopimuksen voimassaoloaikaa.

Elätys

Elätyksellä tarkoitetaan tässä työssä vakuutussopimusten elinkaaren elättämistä automaation avulla halutun määrän päiviä eteenpäin. Elätys simuloi vakuutussopimuksen elämistä tietojärjestelmässä ajan kuluessa eteenpäin.

MVC

MVC on lyhenne sanoista *model-view-controller*, joka suomennettuna tarkoittaa malli-näkymä-käsittelijä -tapaista arkkitehtuurimallia. Mallilla tarkoitetaan järjestelmän tiedon tallennus-, ylläpito- ja käsittelytoimia. Näkymällä tarkoitetaan graafista käyttöliittymää tietojen esittämistä varten. Kontrollerilla tarkoitetaan käsittelijää eli ohjainta, joka vastaanottaa käyttäjältä käskyjä ja tekee ohjelmoituja toimintoja niiden perusteelta.

Testauspainotteinen kehittäminen

Kehittämistapa, jossa vaatimusten pohjalta suunnitellaan ensin testitapaukset lähdekoodin ominaisuuksien testaamiseen. Suunniteltujen testien pohjalta rakennetaan toteutettava lähdekoodi.

Testiautomaatio

On ohjelmisto, joka automatisoi testausta. Testiautomaatio voi koostua yhdestä tai useammasta osasta, joiden avulla testataan toista tietojärjestelmää.

Sisällys

1 Johdanto

Työ toteutettiin toimeksiantona Sp-Henkivakuutus Oy:lle. Yritys on aloittanut toimintansa vuonna 2006 toiminimellä Henkivakuutusosakeyhtiö Duo. Uusi toiminimi otettiin käyttöön 18.8.2014. Liiketoiminta perustuu henki-, säästö- ja eläkevakuutuspalvelujen sekä lainaturvan, kapitalisaatiosopimuksien ja varainhoitovakuutuksen tuottamiseen Säästöpankkien henkilö- ja yritysasiakkaille.

Keskeisimpinä haasteina yrityksen toiminnassa on kehittää sellaisia tietoteknisiä ratkaisuja, joiden avulla tavoitetaan asetetut liiketoiminnalliset tavoitteet. Eräs tapa toteuttaa tavoite on luoda liiketoiminnan vaatimuksiin perustuva tietojärjestelmä, jossa toimeksiantajalla keskeiseksi toiminnaksi muodostuu vakuutussopimusten hallinta. Tietojärjestelmän jokapäiväisen käytön lisäksi tulee osana myös ylläpidolliset tehtävät sekä kehittämistyöt, jotka kuormittavat käytettävissä olevia resursseja. Uuden toiminnollisuuden kohdalla laatu ei saa kärsiä. Jotta varmistetaan, että nykyisillä resursseilla voidaan taata tietojärjestelmään tarpeellinen laadukkuus, tarvitaan tämän tuottamiseksi resursseja säästävä tapa testata ja validoida tietojärjestelmää.

Tämän opinnäyteprojektin tavoitteena on tuottaa toimeksiantajalle toimiva vakuutussopimusten elinkaarta testaavan ohjelma käytössä olevaan tietojärjestelmään. Ohjelman tulee tukee liiketoiminnallisia intressejä sekä edistää hyväksyntätestauksen laatua automatisoinnin avulla. Ohjelmiston keskeisempänä tavoitteena on tuottaa viiteaineistoa, jota tietojärjestelmä pystyy käsittelemään. Toiseksi ohjelmiston tulee pystyä luomaan satunnaistapahtumia. Kolmanneksi tapahtumien luontiin pitää luoda eräänlainen lukitusmekanismi, joka lukitsee vakuutussopimuksia siten, ettei koko kantaan suoriteta muutosajoa kerralla. Lisäksi jokaisesta tapahtumasta pitää pystyä kirjoittamaan niin selkeä päiväkirja, jotta vastaavanlainen tapaus voitaisiin manuaalisesti pystyä suorittamaan toistettavuuden vuoksi.

Ohjelmiston automatisoidun elinkaaritestauksen avulla voidaan vakuutussopimusten elinkaarta simuloida tietojärjestelmässä lyhemmällä aikavälillä. Lyhempi aikaväli perustuu siihen, ettei tarvitse odottaa päivittäin muodostuvia aineistoja, vaan aineisto luodaan automatisoinnin avulla vastaamaan tuotantoympäristössä vastaavankaltaisia aineistoja. Vakuutussopimusten elinkaaritestauksen avulla voidaan saada tarkempi kuva, kuinka vakuutussopimukset käyttäytyvät tietojärjestelmässä. Käytännössä elinkaaritestaus perustuu viiteaineistojen tuottamiseen, käsittelyyn, tulosten verifiointiin käyttöliittymästä sekä muodostuvien aineistojen automaattiseen oikeellisuuden tarkistukseen liiketoiminnallisten määritysten mukaisesti.

Liiketoiminnallisen näkökulman kautta projekti edesauttaa käytettävän tietojärjestelmän laadunvarmistamista sekä vakuutustuotteiden testausta. Automatisoinnilla voidaan samalla vähentää testauskuluja, koska manuaalista testaamista voidaan kohdentaa tarkemmin erityistapauksien testaamiseen.

Seuraavassa luvussa käsitellään laadun ja automatisoinnin määritelmiä. Kolmannessa luvussa kerrotaan projektinhallinnallisista menetelmistä, joilla opinnäyteprojektia hallittiin. Neljäs luku sisältää ohjelmiston kehittämiseen sovelletun teorian liittymisen keskeiseen kehitysmenetelmään, DevOpsiin. Ohjelmistoratkaisun kuvaus ja siihen liittyvä tekninen tausta on esitettynä viidennessä luvussa. Kuudes luku keskittyy kehitystyöhön, jossa varsinainen ohjelmistokehitystyö toteutettiin. Seitsemäs luku analysoi ja pohtii saavutettuja tuloksia. Viimeinen luku sisältää keskeisimpiä havaintoja ja pohdintoja, kuinka projektissa onnistuttiin valittujen käytänteiden osalta ja kuinka ne vaikuttivat kehitystyöhön. Jatkokehitysideoita on kerätty viimeiseen lukuun.

Työhön ei ole liitetty salassa pidettävää aineistoa, muuta kuin kuva 6, jossa tekstit ovat sensuroitu. Lähdekoodiesimerkit eivät suoraan liity tuotettuun ratkaisuun. Sen sijaan lähdekoodiesimerkit esittävät vastaavia ideoita ja teemoja.

2 Laatu ja automatisointi

Laatu ja automatisointi ovat käsitteitä, jotka riippuvat siitä, mistä näkökulmasta asiaa tutkitaan. Yleisellä tasolla molemmat ovat määriteltävissä, mutta teollisuustuotannossa automatisointi ja laatu eivät välttämättä ole täysin samankaltaisia kuin tietoteknisellä alalla.

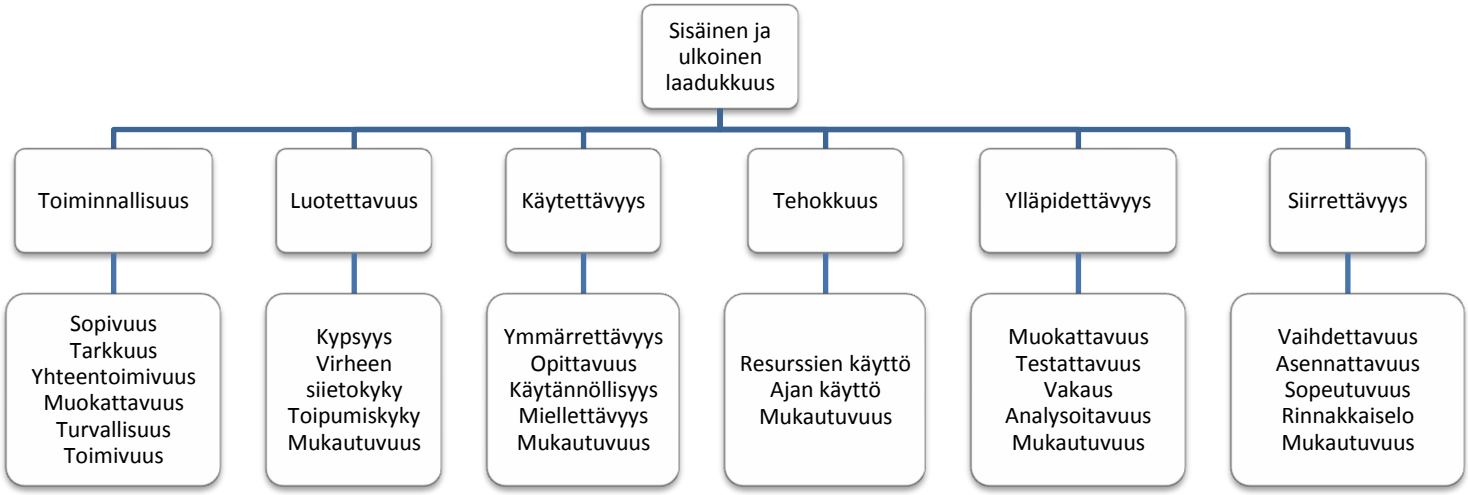
Seuraavaksi käsitellään sekä yleisellä että tietoteknisellä tasolla laatua ja automatisointia. Tunnistamalla keskeiset laadukkuuden tekijät, tunnistetaan ne osa-alueet, joista laatu muodostuu.

2.1 Laatu

Juranin ym. (1999, 26-27) mukaan laadulla voidaan tarkoittaa tuotteen niitä ominaisuuksia, jotka vastaavat asiakkaan tarpeita ja tuottavat asiakastyytyväisyyttä. Saavuttaakseen korkean laadun useimmiten laadusta maksetaan enemmän. Toisaalta laadulla voidaan tarkoittaa myös vapautta puutteista, jolloin virheitä, uudelleen työstämistä tai asiakastyytymättömyyttä ei esiinny. Saavutettu korkea laatu on maksanut vähemmän. Erona näissä kahdessa on näkökulma. Ensimmäisessä tarkastellaan tilannetta myynnin kannalta, jälkimmäisessä näkökulmana tarkastelee laatua kustannuksien kannalta.

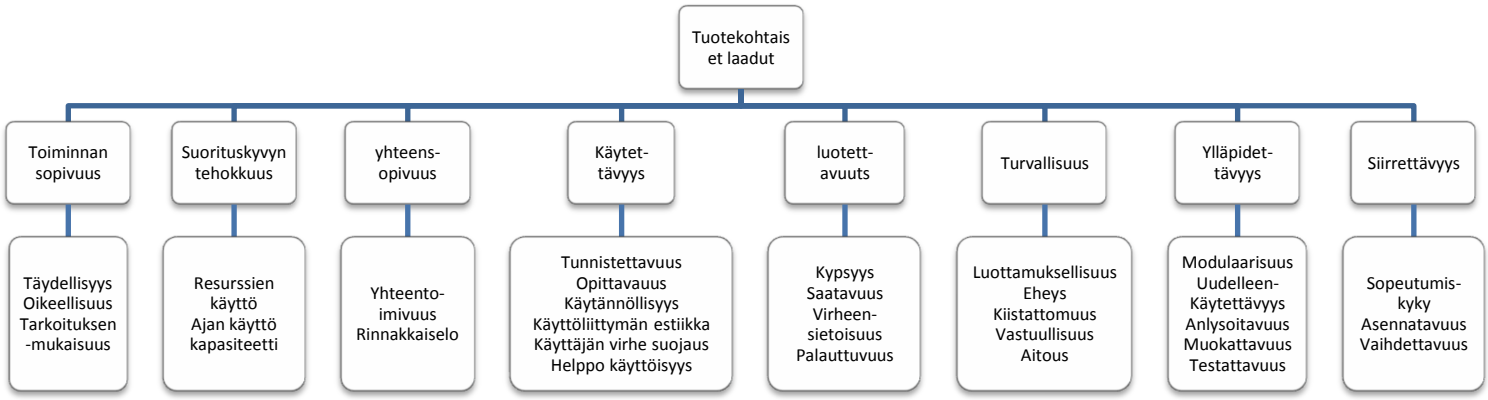
Laatua voisi luonnehtia monella tapaa ymmärrettävä termiksi. Laatu riippuu käsiteltävästä kontekstista. Yksinkertaisuudessa tämä käsittää asiakkaan odotuksien ja vaatimusten toteutumista sekä sille määritettyjen vaatimusten yhteensopivuutta ja sopusointua (Lipponen 1993, 218).

Ohjelmistojen laadukkuudelle on määritetty yleisiä standardeja, jotka tunnetaan ISO-standardeina. ISO-9126-1 (2000)–standardissa laatu on määritetty ohjelmiston sisäisinä ja ulkoisina ominaisuuksina, kuten kuvassa 1 esitetyt kuusi pääkategoriaa: toiminnallisuus, luotettavuus, käytettävyys, tehokkuus, ylläpidettävyys ja siirrettävyys. Jokainen niistä jakaantuu vielä alaosiin täydentämään pääkategoriaa.



Kuva 1. Sisäinen ja ulkoinen laadukkuus ISO-9126 mukaan

ISO/IEC 25010 (2011) –standardi tarkastelee erikseen tuotelaatua ja tuotteen käytön laatua. Jälkimmäistä määrittelevät laatutekijät liittyvät ohjelmiston käyttämiseen ja jakaantuvat viiteen pääkohtaan. Ne ovat vaikuttavuus, tehokkuus, tyytyväisyys, riskittömyys ja kattavuus. Tuotekohtaiset laadut jaetaan kuvan 2 mukaisesti kahdeksaan osaan: toiminnan sopivuus, suorituskyvyn tehokkuus, yhteensopivuus, käytettävyys, luotettavuus, turvallisuus, ylläpidettävyys ja siirrettävyys. Nämä jaetaan myös pienempiin osa-alueisiin edellisen standardiversion mukaisesti.



Kuva 2. Tuotekohtaiset laadut ISO/IEC 25010 mukaan

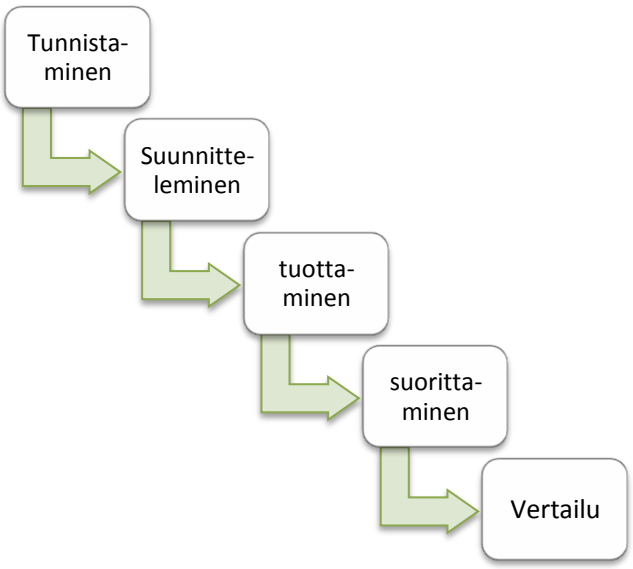
Tuntemalla tuotteen laatutekijät, voidaan tunnistaa tuotteesta ne kohdat, joissa yksittäisille laatutekijöille voidaan antaa mitattava arvo. Näitä mitattavia ominaisuuksia voidaan tarkastella objektiivisesti ja suhteuttaa tuotteen saavutettuun arvoon. Nämä yhdessä muodostavat tuotteelle ominaisen laadun. (Lipponen 1993, 35.)

2.2 Automatisointi

Sanakirjan määritelmänä automaatiolla tarkoitetaan itsenäisesti toimivien koneiden ja laitteiden käyttämistä sellaisiin työtehtäviin, jotka muuten vaativat ihmiseltä usein samankaltaisia toimenpiteitä (WSOY 2006, 63). Automatisoinnilla tarkoitetaan, että usein samankaltaisen työn toimenpiteet muutetaan tietokoneella ohjelmoiduksi toteutukseksi, joka vastaa käsin suoritettavaa työsuoritusta (Mansio 2015).

Testiautomaatiolla tarkoitetaan sellaisia toimia ja ponnisteluja, joiden tarkoituksena on automatisoida testattavia tietoteknisiä prosesseja ja operaatioita. Gaon ym. (2003, 157-159) mukaan nämä saavutetaan käyttämällä hyvin määritettyjä strategioita sekä systemaattisia ratkaisuja. Tällöin laatu voidaan mitata, kuten Black ym. (2009) toteavat, tietojärjestelmässä esimerkiksi löydettyjen vikojen ja testien määrällä sekä suoritettujen testien testikattavuudella.

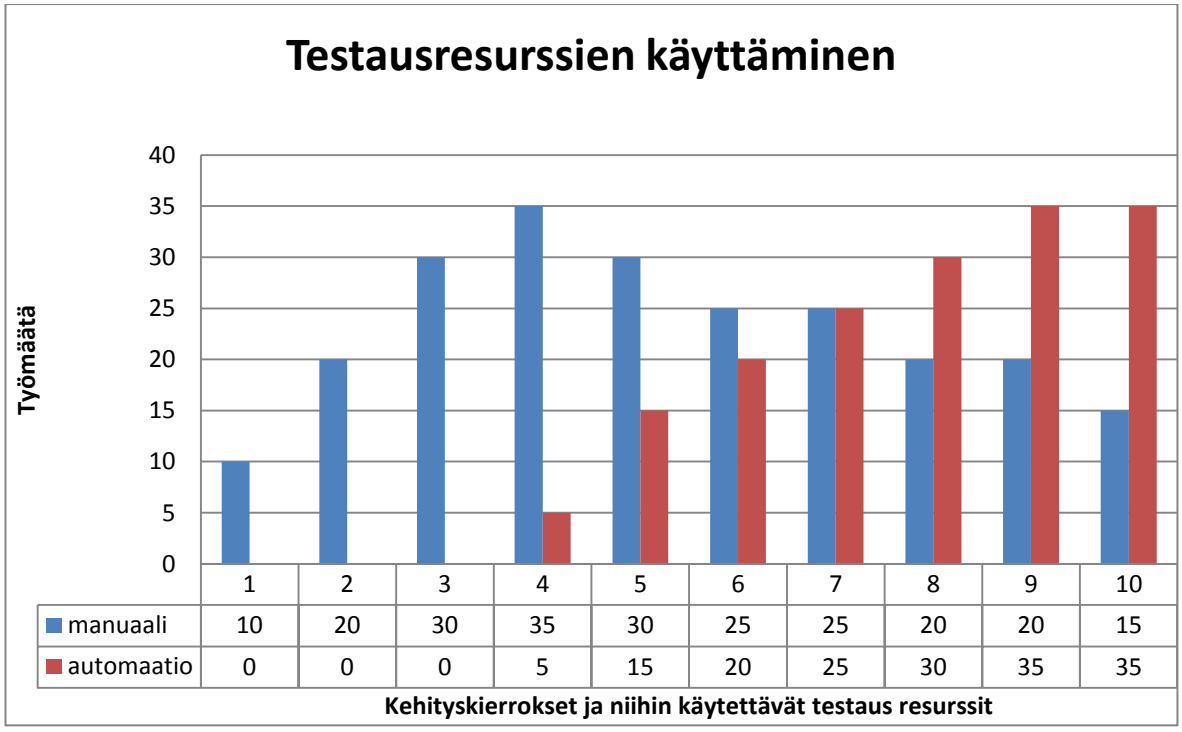
Testiautomaatiossa, ja yleensä testauksessa, testit rakentuvat testattavan tai testattavien toiminnollisuuksien testaamiseen testitapauksilla. Testiautomaatiossa näitä testitapauksia ajetaan automaattisesti. Yksittäinen testitapaus voi koostua monista eri vaiheista ja ennakkoehdoista, jotka voivat vaikuttaa testitapauksen suunnitteluun ennen testin tuottamista. Testitapauksen suorittamiselle on odotettavissa vaatimusten mukainen lopputulos. (Fewster & Graham 1999, 13-22). Kuvassa 3 on kuvattuna testitapauksen suunnitteluprosessi, joka koostuu viidestä vaiheesta: tunnistamisesta, suunnitellusta, tuottamisesta, suorittamisesta sekä tuloksen vertailusta odotettuun lopputulokseen.



Kuva 3. Testitapauksen suunnitteluprosessi

Fewsterin ja Grahamin (1999, 65-100) mukaan testiautomaattiin kuuluu oleellinen osa tuottaa testiaineistoja, joiden avulla testausta voidaan suorittaa. Aineiston muodostamiseen liittyy siihen liittyvät määritykset, jotka riippuvat testattavasta ohjelmasta. Muodostaminen voi tapahtua ohjelmoitujen tapahtumien kautta, jolloin aineisto muodostetaan sen määritysten mukaisesti.

Mansion (2015) mukaan keskeisempiä etuja testiautomaatiossa ovat manuaalisen testauksen kohdentaminen luovuutta vaativiin testitapauksiin, toistuvien ja aikaa vievien testitapausten ajamiseen koneen avulla sekä testikattavuuden parantaminen. Fewster ja Graham (1999, 3-10) lisäävät edellä mainittuihin hyötyihin toistettavuuden, usein ajettavuuden, taloudelliset säästöt sekä testien uusiokäyttämisen toisen testin osana. Lisäksi ohjelmiston laadun parannusta saavutetaan liittämällä automatisoidut testit manuaalisien testien rinnalle. Kuvitteellisessa kuvassa 4 on kuvattu kuinka automaatiolla voidaan vaikuttaa testiresurssien käyttämiseen. Kuvassa neljännessä vaiheessa käyttöön otettu automatisointi on vähentänyt manuaalisen testaukseen käytettäviä testausresursseja.



Kuva 4. Testiresurssien käyttäminen

Toisaalta sekä Mansio (2015) että Fewster ja Graham (1999, 22-25) toteavat että automatisointi ei ole mahdollista toteuttaa kaikkiin testitapauksiin. Myöskään manuaalista testausta ei saisi unohtaa, sillä automatisoidut testit eivät luonteeltaan ole muuttuvia. Automatisointi tulisi toteuttaa vain niille testitapauksille, jotka eivät ole liian haastavia ja aikaa vieviä toteuttaa sekä sellaiset tapaukset, jotka ovat ainoastaan todennettavissa fyysisen kontak-

tin kautta. Lisäksi manuaalinen testaaminen tuo enemmän luovuutta esille, jota ohjelmallisesti automatisoitu testiautomaatti ei pysty tuottamaan. Hyvä on myös huomioida, että testiautomaation luominen voi olla kallista toteuttaa sekä ylläpitää, mutta testitapausketjujen ajaminen pitkällä aikavälillä, voi tuottaa säästöjä.

3 Projektin hallinta

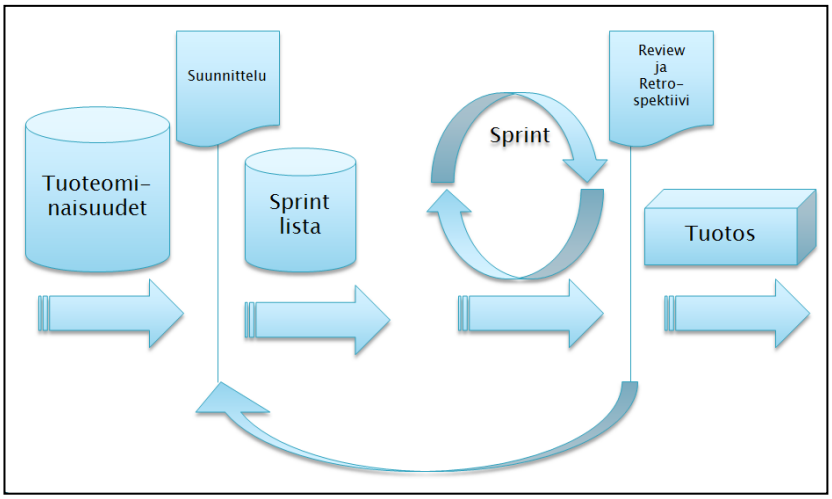
Projektin hallinta on yksi kokonaisuus, jossa halutaan luoda projektille mahdollisuudet onnistua. Näihin kuuluvat muun muassa projektin keskeisimmän metodologian valitseminen sekä tavat ohjata ja hallita projektia.

Seuraavissa kappaleissa kuvataan tarkemmin, mitä projektin hallinnallisia ajattelutapoja työn aikana käytettiin. Samalla avataan keskeisempiä termejä ja prosesseja.

3.1 Scrum ja ketterä ohjaus

Scrumiin kehittäjät Schwaber ja Sutherland (2013, 2-15) määrittelevät scrumin viitekehykseksi, jota käytetään kehitettäessä tuotetta. Scrum ei ole kuitenkaan tuotekehitysprosessi tai tekniikka tuottaa tuotteita, vaan se on projektinhallinnallinen viitekehys, joka tekee tuotekehityksen näkyväksi. Se ei myöskään sido yksittäisiä käytettäviä prosesseja ja tekniikoita itseensä, vaan suosittaa viitteitä näiden hallinnoimiseen yhteisen sanaston ja sääntöjen avulla, jolloin scrumiin voidaan liittää halutessa joukko muita prosesseja ja tekniikkoja.

Schwaber ja Sutherland (2013, 2-15) täydentävät, että viitekehys tarjoaa projektille erilaisia roolituksia ja käsitteitä, kuten tuotteen kehitysjonon, iteratiivisen kehityskierroksen määritelmän *(sprintin)*, edistymisen seuraamisen, tuoteversion sekä kehitysprosessin tarkastelun *(retrospektiivin).* Lisäksi viitekehys tarjoaa muodollisia tapahtumia, joiden aikana voidaan katselmoida ja suunnitella tulevaa. Tärkeimmät näistä ovat sprintin suunnittelu, päiväpalaverit, sprintin katselmoinnit sekä katselmoinnin jälkeinen retrospektiivi. Kuvassa 5 on hahmotettuna scrum-prosessi.



Kuva 5. Scrum-prosessi

Scurimissa kehittäminen tapahtuu iterointikierroksissa, joita kutsutaan *sprinteiksi*. Tuoteomistaja on nimetty henkilö, joka vastaa kokonaan tuotteen kehitysjonosta. Tähän kuuluu tuotteen kehitysjonon selkeys, ymmärrettävyys, järjestys sekä toiminnolliset vaatimukset. Tuoteomistaja hyväksyy työn valmistumisen katselmoinnissa, mikäli se hänen mielestään täyttää valmiin määritelmän. Lisäksi tuoteomistajan tulee tiedostaa kehitysjonon tilanne ja sen mukaan arvioida valmistuminen. Tuoteomistajan tuottaman tuotekehitysjonon toteuttaa itseohjautuva kehitystiimi, jossa jokainen kehittäjä on vastuussa tuotteen kehitysjonon edistymisestä. Kehitystiimin sisällä ei ole alitiimejä ja tehokas tiimi koostuu kolmesta yhdeksään kehittäjään. Kehitystyötä valvoo scrummaster, jonka pääasiallisena tehtävä on huolehtia scrumin toteutumisesta. Tämä koskee sekä kehittäjien että tuoteomistajan puolta. Tämä tarkoittaa muun muassa tapahtumien järjestämistä, aikaraameissa pysymisestä sekä tuottaa koko projektin aikana läpinäkyvyyttä sekä kehitysorganisaation että tuoteomistajan puolelta. (Scrum Alliance 2014).

Kehityskierros alkaa sprintin suunnittelulla, jossa päätetään tuoteomistajan, kehitystiimin sekä scrummasterin kanssa sprintin yhteinen tavoite. Tavoitteessa eritellään mitä tullaan toteuttamaan, toteutuksen arvioidun tuloksen sekä suunnitelman tämän toteuttamiseksi. Sprintin kesto on enintään yksi kalenterikuukausi, jonka aikana järjestetty sprintin kehitysjono toteutetaan. (Scrum Alliance 2014.)

Sprintin kehitysjono muodostuu tuotekehitysjonosta, josta kehitystiimin ja tuoteomistajan yhtenäisen ymmärryksen nimissä otetaan toteutettava määrä työtä sprintille. Sprintin aikatavoitetta tarkastellaan lyhyissä päiväpalavereissa, jotka toteutetaan kehitystiimin puolesta tietyssä paikassa tiettyyn aikaan. Palaverissa selvitetään mitä tehtiin, mitä tullaan tekemään ja onko työnteolle esteitä. (Scrum Alliance 2014).

Sprintti päättyy katselmointiin, jossa kehitystiimi esittelee sprintin tuloksen tuoteomistajalle sekä tuoteomistajan kutsumille henkilöille. Katselmoinnissa tuoteomistaja päättää mitkä tuotekehitysjonon osat ovat valmistuneet ja esittää arvionsa valmistumisajankohdasta perustuen edistymiseen. Lisäksi tarkistetaan projektin aikataulu, budjetti sekä pohditaan mitä seuraavassa vaiheessa tulisi kehittää. Katselmoinnista seuraa välittömästä kehitystiimin retrospektiivi, jossa kehitystiimi keskustelee missä onnistuttiin ja missä on parantamisesta. Aiheet voivat liittyä kehitysprosessiin, yhteistyöhön ja työkaluihin. Retrospektiivin tuloksena syntyy suunnitelma keskeisempien asioiden parantamiseksi sekä mahdollisen valmiin määritelmän sopeuttamisesta. Retrospektiivin jälkeen kehityskierros seuraavan sprintin suunnittelulla, kunnes tuotteenkehitysjono on tyhjentynyt. (Scrum Alliance 2014.)

Erityishuomiona Schwaber ja Sutherland (2013, 12-15) korostavat, että tuotekehitysjono ei koskaan alussa ole täysin valmis, koska kaikkia toiminnollisuuksia ja ympäristöjen tuomia ominaisuuksia ei tunnisteta. Järjestetyssä tuotekehitysjonossa myös korkeammalle priorisoiduimmat tuoteominaisuudet ovat yleensä selkeämmin kuvattuja ja luontevampia ottaa kehitykseen kuin alempana olevat, koska niiden arvo ja yksityiskohtaisuus on selkeämmin määritetty.

3.2 Lean, kanban ja arvon tuottaminen

Isomäen ym. (2014, 8) mukaan Lean on johtamisoppi, jonka tarkoituksena on minimoida lisäarvoa tuottamaton työ ja jatkuva toiminnan kehittäminen. Yksi näistä tavoista on optimoida arvoketjua kanban-menetelmän mukaan, jossa työvuota seurataan esimerkiksi visuaalisen taulun avulla.

Vuori (2010, 2) määrittelee Leanin perusajatuksen näkemykseksi kevyestä toimintatavasta. Ohjelmistokehitykseen Lean liittyy Mary ja Tom Poppendieckin (2007, 23-41) mukaan seitsemän kokonaisuuden kautta. Näitä ovat: hukkien eliminoiminen, laadukkuus tuotteen osana, tietämyksen kasvattaminen, sitoutumisten lykkäämiseen, nopeat toimitukset, työyhteisön kunnioittaminen sekä kokonaisuuden optimointi.

Isomäen ym. (2014, 8) mukaan yksinkertaisin tapa eliminoida hukkia on olla tekemättä ominaisuuksia, joita ei tarvita. Poppendieckit (2007, 23-25) tarkentavat asiaa määrittelemällä tarkemmin hukaksi kaiken sen, mikä ei lisää tuotteelle arvoa. Näitä ovat muun muassa pitkät vaatimuslistat, osittain tehty työ ja niin sanottu kirnuaminen, joka johtuu muuttuneista vaatimuksista. Koch (2004, 253) vuorostaan määrittelee ratkaisuksi näihin hukkien tunnistamisen ongelmiin arvoketjun tunnistamisen osana kehitysprosessia.

Poppendieckit (2007, 25-29) asettavat tavoitteeksi rakentaa laadukkuus osaksi tuotetta siten, ettei vikoja syntyisi. Ihannetilanteessa vikojen seurantajärjestelmää ei tarvita. Tämä on saavutettavissa testauspainotteisella kehittämisellä käyttäen hyväksi integrointijärjestelmiä. Uudelleen työstämisen vähentämiseksi sekä laadun parantamiseksi Alan (2008, 25-26) listaa seuraavia käytänteitä: lähdekoodin katselmointi, pariohjelmointi, testiautomaatiot, testauspainotteinen kehittäminen, jatkuva integrointi sekä joustavat mallit.

Lillrank ym. (2003, 16-22) mukaan mikäli laadukkuus saadaan rakennettua osaksi tuotetta ja sen laatukustannukset voivat vähentyä. Laatulaskennassa laatukustannuksiksi lasketaan kaikki ne kustannukset, mikäli kaikki tehtäisiin kerralla oikein. Samalla nämä kustan-

nukset kertovat sen, kuinka paljon lisäarvoa tuottamatonta työtä on syntynyt kehitystyön aikana.

Koch (2004, 254) kertoo tietämyksen kasvamisen neljän seikan avulla. Ensimmäisenä hän mainitsee palautteen saamisen. Palaute voi tulla testaamisen, prototyyppien sekä kehitteillä olevan järjestelmän esittämisestä muille. Toisena hän mainitsee iterointiin perustuvan kehittämisen, kolmannen muodostavat ohjelman muutokset ja neljäntenä sarjoihin perustuva kehitys, jossa vaikeaan ongelmaan kehitetään muutamia mahdollisia ratkaisuja ennen ratkaisujen yhdistämistä. Poppendieckit (2007, 29-32) lisäävät mukaan kehitystyön aikana tapahtuvaa oppimista, joka parhaimmillaan parantaa myös kehitystyötä kokemusten kautta. Waters (2010) ynnää tietämyksen esittämistavat muun muassa dokumentoinnin, wikien ja lähdekoodin kommentoinnin avulla. Myös pariohjelmointi, harjoittelulla ja lähdekoodin tarkastelulla voidaan lisätä tietämystä.

Poppendieckit (2007, 32-33) esittävät sitouttamisen lykkäämisen siten, että päätökset tulisi tehdä mahdollisimman myöhään sallitun aikarajan sisällä. Tällöin tietämys on kasvanut suurimmilleen ja vaihtoehtoisia ratkaisuja on ehditty kokeilemaan, joiden avulla voidaan päätös tehdä, ja peruuttamattomat muutokset tulisi aikatauluttaa mahdollisimman myöhäksi.

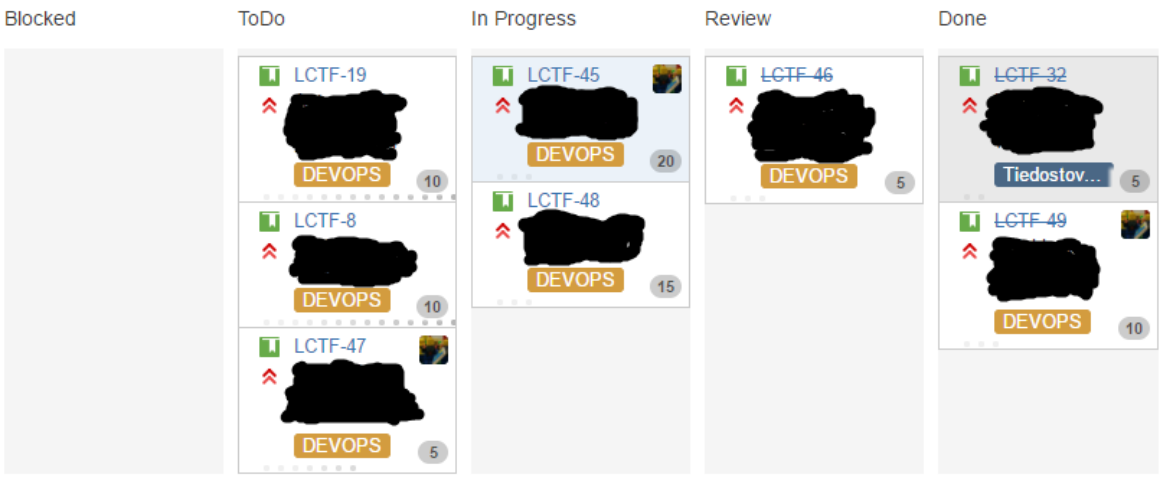
Iteroinnin avulla Hibbs ym. (2009, 21-22) mukaan on mahdollista suorittaa nopeita toimituksia asiakkaalle. Asiakkaalla on tilaisuus nähdä toimintoja, saada ja antaa palautetta, joka voi aiheuttaa muutoksen toisessa toiminnallisuudessa ennen sen toteuttamista. Lisäksi eliminoidaan uudelleen työstäminen sekä varmistetaan että toiminnallisuus vastaa asiakkaan tarvetta. Tähän Poppendieckit (2007, 34-35) lisäävät, että nopeilla toimituksilla on myös mahdollista saavuttaa kehitystyössä etuja. Esimerkiksi uusia ideoita voidaan kokeilla ja niistä saadaan uutta tietämystä.

Työyhteisön kunnioittamisella Poppendieckit (2007, 36-38) tarkoittavat yrityksen sisäisiä käytänteitä, jotka he jakavat kolmeen pääkohtaan. Aloitekykyisillä johtajilla yritys varmistaa, että tiimeillä on mahdollisimman hyvät johtamistaidot, jotta tiimeistä voi kehittyä sitoutuneita ja omatoimimisia. Silloin voimavarat keskittyvät hyvien tuotteiden kehittämiseen. Toiseksi tulee vaalia ja kehittää teknistä asiantuntemusta tiimeissä, jotta jatkuva kehittyminen varmennetaan. Kolmanneksi tiimeille tulee antaa vastuuta siten, että he pystyvät suoriutumaan työstä oman osaamisen kautta, eikä tiimille tarvitse erikseen kertoa, kuinka työ- ja tulostavoitteet saavutetaan. Koch (2004, 255) korostaa, että motivoituneet tiimit toimivat paremmin. Tällöin tiimi jakaa yhteisen päämäärän, ylläpitää me-henkeä, rakentaa

luottamuksen ja turvallisuuden ilmapiiriä sekä ymmärtää edistymisen seurannan merkityksen.

Kokonaisuuden optimointiin liittyy arvoketjun määritelmä. Arvoketjulla tarkoitetaan kaikkia niitä toimintoja ja prosesseja, jotka lisäävät tuotteelle arvoa. Arvoketjun rakentaminen lähtee ruohonjuuritason raa'asta materiaalista, josta tuotetaan tuote loppukäyttäjälle (Rother & Shook 2003, 3). Kokonaisuuden optimoinnilla Hibbs ym. (2009, 21-22) tarkoittavat koko arvoketjun parantamista, ei vain keskittymistä arvoketjun tiettyyn osaan. Tällöin syntyy alioptimointia, jolloin arvoketju ei välttämättä parane.

Kanban-menetelmällä Hobbsin (2014, 28-37) mukaan voidaan saavuttaa työvuon seurantamenetelmä, jossa optimoidaan työtaakkaa valmistumisvaiheiden välillä. Menetelmän avulla voidaan seurata yksittäisen työn sekä koko työnmäärän edistymistä. Tämän avulla voidaan tunnistaa valmistumisprosessi paremmin, jolloin työn alla olevat tehtävät tunnistetaan paremmin. Kanbanin avulla pyritään ehkäisemään työn pakkaantumista työvaiheiden väliin. Ketterään ohjelmistokehitykseen kanban saadaan Saddingtonin (2012, 131-135) mukaan tehtävienhallinnan avulla tunnistamalla tehtävän tila visuaalisen taulun avulla. Kuvan 6 taulussa tehtävää siirretään nimettyjen uimaratojen välillä, jolloin tiedostetaan nopeasti tehtävien tilanne kunkin toiminnollisuuden osalta. Salassapitovelvollisuuden vuoksi tehtävien nimiä ei voida näyttää.



Kuva 6. Kanban-taulu tehtävienhallintaan

Cooke (54-56) muistuttaa että kanban asettaa rajoja työn alla oleviin tehtäviin. Kehitystiimin tulisi ottaa työn alle vain sen verran, kuin sen ajatellaan pystyvän tekemään kehityskierroksen aikana. Lisäksi kanban ei täydellisenä sovi ohjelmistokehitykseen aikarajoitetun kehityskierroksen vuoksi, mutta sen tuomia ideoita tehtävienhallinnasta voidaan soveltaa.

3.3 Ketterä kehitysprosessi

Ketterä ohjelmistokehitys määrittyy seuraavan manifestin mukaisesti. Siihen liittyy myös manifestin pohjalta luodut 12 pääperiaatetta (Liite 1), jotka tarkentavat manifestin sisältöä.

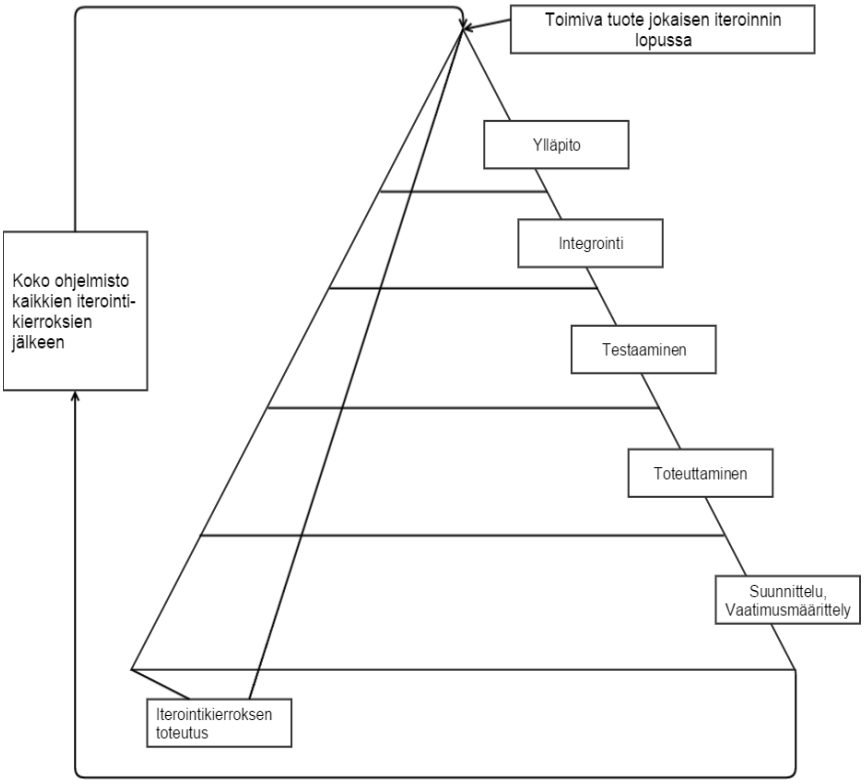
> Löydämme parempia tapoja tehdä ohjelmistokehitystä, kun teemme sitä itse ja autamme muita siinä. Kokemuksemme perusteella arvostamme:
>
> - Yksilöitä ja kanssakäymistä enemmän kuin menetelmiä ja työkaluja
> - Toimivaa ohjelmistoa enemmän kuin kattavaa dokumentaatiota
> - Asiakasyhteistyötä enemmän kuin sopimusneuvotteluja
> - Vastaamista muutokseen enemmän kuin pitäytymistä suunnitelmassa
>
> Jälkimmäisilläkin asioilla on arvoa, mutta arvostamme ensiksi mainittuja enemmän. (Beck ym. 2001).

Goodpasture (2014, 4-5) selventää kirjassaan *Project Management the Agile Way* manifestia. Vaikka menetelmät ja työkalut tarjoavat projektille viitekehyksen, tulisi kanssakäymistä ja yksilöitä huomioida enemmän, jolloin kehittämistä ei suoriteta teknologia edellä. Dokumentaatio tuottaa arvoa sovellukselle, mutta toimiva ohjelmisto tuottaa vielä enemmän arvoa. Asiakasyhteistyössä asiakas tulisi tuoda lähemmäksi kehitystä, eikä pitää vain neuvottelukumppanina. Muutosta tulisi arvostaa asiakastyytyväisyyden kautta enemmän kuin suunniteltua suunnitelmaa, koska tällöin voidaan tarvittaessa toimia dynaamisemmin.

Cooken (2012, 32-43) mukaan ketterät kehitysmenetelmät tarjoavat ratkaisuja perinteisiin ohjelmistokehitys ongelmiin. Näitä ovat muun muassa järjestelmien ylisuunnittelua, viestintäyhteyden katoaminen liiketoiminnan ja kehittäjien välillä sekä vesiputousmallista kehittäminen, jossa seuraavaan vaiheeseen siirtyminen edellyttää edellisen vaiheen täydellistä totuttamista. On myös tavanomaista, että liiketoiminta näkee ensimmäisen kerran kehitetyn ohjelmiston, kun kehitystyö on saavutettu loppuun. Niinpä kehitetyssä tuotteessa voi olla liiketoiminnan kannalta käyttämättömiä toiminnollisuuksia, suunnitteluvirheitä eikä toiminnollisuudet välttämättä vastaa liiketoiminnan tarpeita. Ketterissä menetelmissä pyritään löytämään ratkaisuja näihin ongelmiin muun muassa inkrementaalisen kehittämisen kautta, lyhyillä toimituksilla ja vastaamalla muuttuviin toiminnollisuuksiin. Poimala ja Tolvanen (2013) listaavat vielä hyödyiksi muun muassa tiiviin yhteistyön kehittäjien ja liiketoiminnan välillä, toiminnollisuuksien priorisoinnin ja itseohjautuvuuden.

Poimalan ja Tolvasen (2013) mukaan ketterä ohjelmistokehitys tarjoaa mahdollisuuden inkrementaalisen kehittämisen, jossa kehittäminen tapahtuu jatkuvasti parantaen useassa osassa. Varmistus tapahtuu käymällä läpi kaikki ohjelmiston tuottamisen osa-alueet sekä

kehittämistyönvaiheet useaan kertaan. Leffingwell (2008, 123-137) täydentää vielä, että Iteroinnin jokaisessa osassa pyritään saamaan toimiva, testattu ja arvoa tuottava osakokonaisuus lyhyen aikarajan sisällä valmiiksi. Valmis tuote esitettäisiin iterointikierroksen lopussa asiakkaalle erillisessä näyttötilaisuudessa. Kuvassa 7 on hahmotettuna pala palalta kehittäminen, jossa jokainen iterointikierros tähtää valmiiseen tuotteeseen toteuttaen jokaisen ohjelmistokehitysvaiheen. Näitä vaiheita ovat muun muassa ovat suunnittelu, toteuttaminen, testaaminen, integrointi sekä ylläpito. Tuote on valmis, kun pyramidin jokainen pala on kehitetty.



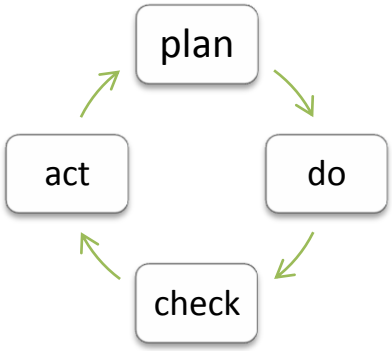
Kuva 7. Ketteräkehitys tähtää toimivaan tuotteeseen jokaisen iteraation lopussa

Kehittämiseen kuuluu myös oleellisesti Cooken (2012, 97-100) mukaan työn seuraaminen ja raportointi. Raportointi ei pääasiallisesti koostu päivittäisestä työajan seurannasta vaan siitä kuinka paljon on onnistuttu tuottamaan ohjelmalle liikearvoa. Raportointi on koko projektin aikana pysyvä aktiviteetti, johon jokainen kehitystiimin jäsen osallistuu. Tärkeimpiä välineitä työn edistymisen seuraamiseen ovat koko projektin vaatimuslista (*requirements backlog*), yksittäisen iterointikierroksen työlista (*delivery backlog*), hallintopaneelit (*executive dashboards*) ja polttokäyräkaaviot (*burndown charts*).

Agile Allicance (2015) määrittelee iterointikierroksen kestävän tietyn ajanjakson verran, useimmiten yhdestä neljään viikkoa, jonka aikana ketterä kehitys tapahtuu. Alussa visioidaan ja suunnitellaan ja lopussa päätetään tehtävät. Leffingwell (2008, 123-127) jatkaa,

että kierrokselle tulisi ottaa tehtäviä sen verran kuin kehitystiimi kykenee suorittamaan. Suunnitteluvaiheessa tehtävät priorisoidaan ja arvioidaan niiden yksilökohtainen työmäärä. Lisäksi päätösvaiheessa tulisi muistaa kehitystiimin sisäinen keskustelu iterointikierrokselta opituista asioista, itse kierroksen reflektointi ja kehitysprosessin pohdinta. Ajatuksena on jatkuva kehittäminen, jolla pyritään vaikuttamaan kehitystyöhön ja tuotteen laadun parantamiseen.

Tämänkaltaista syklistä kehittämistä jatkuvasti parantaen voidaan verrata Demingin kehään (kuva 8), jota kutsutaan myös nimellä PDCA *(plan, do check, act)* tai PDSA (*plan, do, study, act*) malli. The W.Edwards Demingin Instituutin (2016) mukaan mallissa tarkoituksena on jatkuvan parannuksen kautta saadun tiedon avulla parantaen tuotetta tai prosessia. Suunnitteluvaiheessa määritetään päämäärät ja niiden mittaamiseen tekniikat, jotka toteutusvaiheessa toteutetaan. Tarkistusvaiheessa tuotosta testataan ja mitataan, jolloin saadaan tietämystä tuotoksen tilasta ja valmiudesta. Näiden tietojen pohjalta tehdään päätösvaiheessa päätöksiä liittyen tuotokseen, siihen käytetyistä mittaustekniikoista sekä tavoiteasetantaan mahdollisia tarkennuksia tai korjauksia.

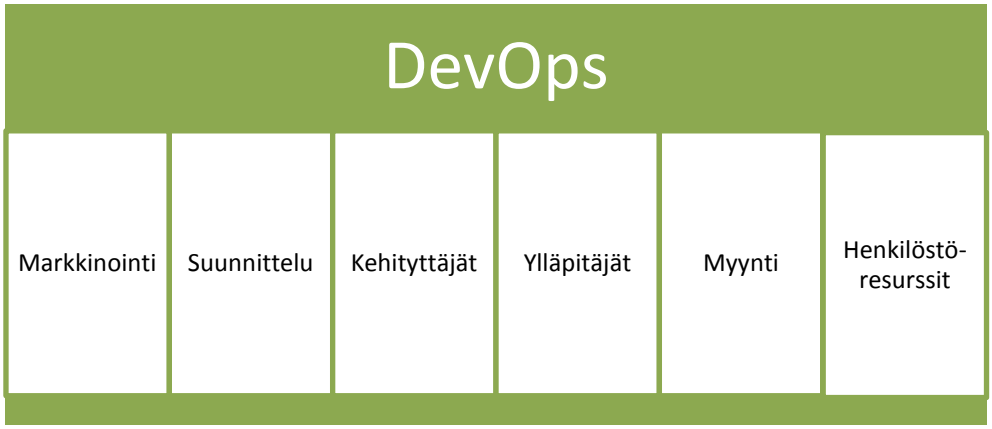


Kuva 8. Demingin kehä

4 Ohjelmiston kehittäminen

Leino ja Thorström (2015) määrittävät Devopsin IT-organisaation toimintatavaksi tehostaa kehittäjien ja ylläpitäjien välistä kommunikaatiota ohjelmistokehityksessä luomalla yhteisen tavoitteen, joka mahdollistetaan nopeilla toimituksilla. Lisäksi suositaan ketterän ohjelmistokehityksen ja lean-ajattelua. Ajatuksena on liittää kehitystyö ja liiketoiminta mahdollisimman lähelle toisiaan. Keskeisimpinä aiheina on jatkuva integraatio ja käyttöönotto, kommunikaatio ja automatisoitavat työvälineet. DevOps vaatii tuekseen yhtenäisen työskentelykulttuurin, prosessit ja työvälineet.

Swartoutin (2012, 33-47) mukaan yhteisen päämäärän visualisoiminen kehittäjien ja ylläpitäjien välille tulee luoda mahdollisimman selväksi, koska eri taustaisille henkilöille asiat voivat tarkoittaa eri asioita. Keinoina tähän Swartout mainitsee yhteisen sanaston käyttämisen, jossa sanasto kirjoitetaan niistä keskeisimmistä termeistä, joita DevOps-rajapinnassa toteutetaan. Esimerkkinä hän mainitsee sanan *julkaiseminen*, joka projektipäällikölle voi tarkoittaa erilaista julkaisemista kuin kehittäjälle. Toisena tapana visualisoida päämäärä on tuottaa organisaatiokuvia kuinka moneen erilliseen liiketoiminnalliseen prosessiin DevOps liittyy. Tällöin DevOps kehittämiseen liittyy oleellisesti liiketoiminnoista markkinointi, suunnittelu, myynti sekä henkilöstöresurssit. Ymmärtämällä DevOpsin liittyvän koko liiketoimintaan pelkästään ohjelmistokehityksen lisäksi voidaan saavuttaa koko liiketoimintaa hyödyttävä kokonaiskuva ja ymmärtää paremmin suhteet. Kuvassa 9 on visuaalisesti selvennetty kyseisiä suhteita Swartoutin mukaan.



Kuva 9. DevOpsin rakentuminen organisaatiossa

Keskeisimmät työvälineet DevOpsissa Swartoutin (2012, 49-70) mukaan ovat versionhallintajärjestelmä, automatisoitu lähdekoodin rakennus- ja testausjärjestelmä, joka toteuttaa jatkuvan integroinnin ja käyttöönoton tuotantoympäristön kaltaisiin ympäristöihin sekä monitorointi- ja mittausjärjestelmä, joka tarjoaa metriikkatietoja. Leino ja Thorström (2015)

lisäävät että työvälineiden avulla julkaisuprosessin tuottaminen on mahdollista. Ketterien menetelmien avulla tuotetaan lähdekoodi ohjelmistoon. DevOps luo prosessin, jossa ohjelmistoon lisätyn ominaisuuden vieminen versionhallintaan mahdollistaa sen katselmoimisen lisäksi automatisoidun integraation ja testausketjun. Automatisoidussa ketjussa ohjelmistoversio asennetaan tuotantoympäristöä vastaavaan ympäristöön automaation avulla. Ympäristössä voidaan suorittaa hyväksymistestaus, jossa suoritetaan integraatio-, regressio-, tietoturva- sekä toiminnalliset ja ei-toiminnalliset testit sekä muut mahdolliset laadunvarmistamismenettelyt. Tuloksen valmistuessa automaation avulla, voidaan julkaisu siirtää hyväksyntään ja viimeisenä siirtää tuotantoon, jolloin sama ohjelmistoversio on käynyt koko julkaisuketjun lävitse.

5 Ohjelmistoratkaisun kuvaus

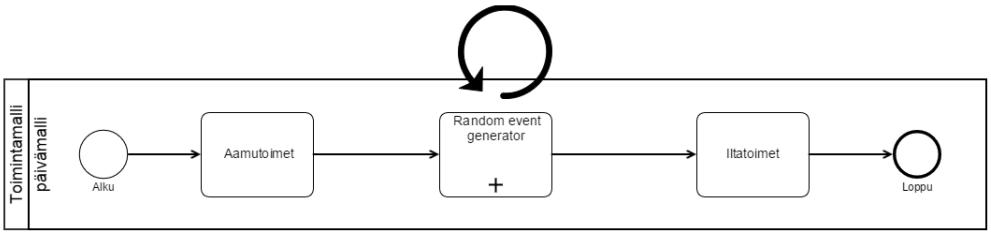
Ohjelmiston elinkaarta testaavan ohjelmiston tavoitteena on testata systeemitestauksen kautta. Black ym. (2009, 43-46) määrittävät systeemitestauksen tarkastelemaan koko sovelluksen toimintaa perustuen sen käyttötapauksiin, vaatimusmäärityksiin ja liiketoiminnallisiin prosesseihin. Lisäksi suhteet ja toiminnot muihin järjestelmiin sekä järjestelmäresursseihin kuuluvat testauksen piiriin. Systeemitestaus toimii osana laajempaa hyväksymistestausta, jonka päämääränä on tarkastella toiminnallisia ja ei-toiminnallisia tuoteominaisuuksia ja niiden laatutekijöitä. Yhteenvetona voidaan määrittää vastaako tuote sille asetettuja vaatimuksia. Toisin sanoen, hyväksymistestauksessa tarkastellaan onko järjestelmä tarkoituksena mukainen.

Ohjelmiston kehittämistyöhön valikoitui DevOps-tyylinen kehitystapa, jossa pystyttiin hyödyntämään ketteriä menetelmiä sekä lean- ja kanban-ajattelutapoja. Menetelmät tukivat pääasiallisesti tavoiteltavia laatu, automatisointi sekä ketteriä tapoja tuottaa laadukas ja toimiva ohjelmisto. Liiketoiminnallisten hyötyjen yhdistäminen kehitystyöhön oleellisesti mukaan asetti sekä vaatimuksia että mitattavia laatutekijöitä. Liitoksella varmistettiin myös, että ohjelmistoon tulee ainoastaan sellaisia toiminnollisuuksia, joita liiketoiminnollisuudet vaativat.

Tarkoituksena oli rakentaa ohjelmisto, joka tuottaa elinkaaritestaukseen viiteaineistomateriaalia sekä kirjoittaa päiväkirjaa mitä aineistoja ohjelmisto on luonut ja jättänyt luomatta. Ensimmäisenä vaatimuksena oli tuottaa materiaali laadukkaasti, ettei virheellinen viiteaineisto estäisi elinkaaritestauksen suorittamista. Vaatimukset viiteaineistoille tuli toimeksiantajan edustajalta ja ne liittyivät suoraan liiketoiminnallisiin vaatimuksiin. Viiteaineistot ovat testattavan tietojärjestelmän toiminnan kannalta sellaisia, joita tietojärjestelmä käyttää päivittäin. Ohjelmistoratkaisussa parametrisuuden avulla tulisi tuottaa vastaavankaltaiset viiteaineistot, joita voidaan parametrisuuden avulla ohjailla. Laatuvaatimuksista tuotettavan ohjelmistoratkaisun pitäisi pystyä mittaamaan ja verifioimaan testattavan ohjelmiston toiminnollisuuksia, luotettavuutta, tehokkuutta ja siirrettävyyttä, jotka esiintyvät ISO 9126 standardissa. Toisaalta tuotettavan ohjelmistoratkaisun piti myös itse täyttää nämä vaatimukset, jolloin tunnistettiin tuotettavat laatutekijät ohjelmistoratkaisulle.

Toisena vaatimuksena oli rakentaa sovellukseen satunnaistapahtumageneraattori, joka tuottaisi sovellukselle tapahtumia, jotka eivät välttämättä aina esiinny tavanomaisessa päiväkäytännössä. Tämä on esitetty yksinkertaistettuna päivämallina, jossa ohjelman käynnistyessä suoritetaan niin sanotut aamutoimet ja illalla iltatoimet. Satunnaistapahtumat kirjoitetaan näiden välissä. Kuvassa 10 tämä on mallinnettuna. Satunnaistapahtumat

kirjoitetaan päiväkirjaan, jotta niitä voitaisiin tarkastella. Satunnaistapahtumilla on tarkoitus demonstroida sitä satunnaisuutta, joka vastaa liiketoiminnassa tapahtuvaa tapahtumaa joka on luonteeltaan sattumanvarainen. Esimerkki testattavan järjestelmän kannalta voisi olla vakuutussopimuksen ennenaikainen päättyminen suunnitellusta päättymisajanhetkestä. Näitä tapahtumia voi syntyä tai jäädä syntymättä riippuen siitä, voiko tapahtuman esiehto toteutua. Esiehtona voi toimia tietokantakysely, jonka palauttaessa arvoja tapahtuma on mahdollista suorittaa. Lisäksi parametrisuudella tarkastellaan tästä löytyneestä joukosta tiettyjä arvoja, joille tapahtumat tulevat tapahtumaan. Päiväkirjaan nämä tapahtumat kirjataan omina tapahtuminaan.



Kuva 10. Päivämallin yksinkertainen kuvaus

Kolmantena vaatimuksena oli rakentaa satunnaistapahtumien yhteyteen eräänlainen lukitsija, joka lukitsee aineistoja tietokannasta, jolle ei voi päivän aikana tapahtua mitään. Parametrien avulla säännöstellään sitä, kuinka monta tapahtumaa voi tapahtua kerralla. Tällä varmistettiin, että kannassa on aina aineistoa, jota voidaan muokata. Tämä estää koko kannan muuttamisen sellaiseen muotoon, että uusia tapahtumia ei voisi muodostua, koska tapahtumien esiehdot eivät täyttyisi. Huomioitavaa tässä on se, että tietokannan ollessa pieni, parametrisuuden avulla voidaan säännöstellä tapahtuvien tapahtumien määriä. Niinpä tietokannan mahdollisten tapahtumien kasvaessa, voidaan parametria muuttaa, jolloin saadaan enemmän mahdollisia tapahtumia aikaiseksi.

Tuntemalla ohjelmiston laatutekijät saatiin suuntaviivoja, miten tuotettavan ohjelmiston laatua tulisi mitata. Yleisellä tasolla laatua mitattiin ohjelmistotestauksen avulla sekä hyväksyntätestien kautta, jolloin tarkasteltiin muodostuneen aineiston muotoa ja ajettavuutta. Tuotettavan ohjelmiston pitäisi pystyä jatkossa taipumaan muiden mahdollisten toimeksiantajan ohjelmistoihin tarpeen vaatiessa, joka asetti lisävaatimuksia ohjelmistoratkaisun rakenteelle. Näinpä ei voitu rakentaa sellaisia rakenteita, jotka lukitsisivat ohjelmistoratkaisut koskemaan vain yhtä tietojärjestelmää.

Seuraavissa ala-luvuissa on käsitelty ohjelmistoratkaisuun liittyvä teknisiä keinoja, joiden tarkoitus on avata tarkemmin tuotettua ohjelmistoratkaisua. Samalla kuvataan, millaisilla tavoilla ratkaisu on kehitetty ja, mitä ratkaisun luomiseen on tarvittu.

5.1 Ohjelmointikieli

Ohjelmiston kehityskielenä käytettiin Javaa. Esimerkit käsitteistä periytyminen, rajanpinnan toteuttaminen ja reflektio löytyvät liitteestä 2. Liitteessä 3 on esimerkki geneerisestä luokasta.

Holznerin (2001, 5-7) mukaan Java on Sun Microsystems kehittämä alusta riippumaton oliopohjainen ohjelmointikieli. Alun perin Sun Microsysteminsin sisäinen ohjelmointikieli, joka tosin tunnettiin nimellä *Oak* kunnessa 1995 nimitus muuttui Javaksi. Samana vuonna ohjelmointikieli julkaistiin yleiseen käyttöön. Ohjelmointikielessä Java ohjelmat toimivat virtuaalisen Java ympäristön, JVM (Java virtual machine), ajamina ohjelmina, joissa ohjelmoinnin lähdekoodi käännetään tavukoodeiksi, joita JVM tulkitsee.

Holznerin (2001, 158) mukaan Javassa oliopohjainen ohjelmointikieli rakentuu muutaman käsitteen ympärille. Nämä ovat: luokka, olio, attribuutit, metodit ja perinnöllisyys. Vesterholm ja Kyppö (2003, 79-170) kuvaavat yksittäisen luokan määrittelevän olion rakenteen ja sen käyttäytymisen attribuuttien ja metodien kautta. Jokainen luokasta muodostuva olio on ainutlaatuinen ja pääasiassa jokainen olio piilottaa attribuuttinsa toisilta luokilta. Näitä attribuutteja käsittelemään julkisien metodien kautta. Luokalla voi olla luokkakohtaisia piirteitä, joita kutsutaan staattisiksi piirteiksi. Staattiset piirteet eivät edellytä luokkakohtaisen olion olemassaoloa.

Harju ja Juslin (2009, 186-209) kertovat, että Javassa luokka voi periä toisen luokan ominaisuuksia ja metodeja käyttämällä avainsanaa *extends*. Tällöin luokkaa joka periytyy toisesta luokasta, kutsutaan aliluokaksi, jolla on suhde yliluokan kanssa. Yliluokka voi olla abstrakti luokka, josta ei suoraan voida luoda oliota. Metodit voidaan luoda abstrakteiksi, jolloin niistä tiedetään paluuarvot ja metodit. Mikäli metodit luodaan abstrakteiksi, tulee aliluokassa täydentää tiedot toimiviksi metodeiksi. Vesterholm ja Kyppö (2003, 174) täydentävät, että täydentäminen tapahtuu syrjäyttämällä peritty samanniminen metodi aliluokassa.

Kätevää on myös määrittää tarvittaessa luokalle rajapinta (*interface)*. Rajapintaluokilla ei ole omia attribuutteja eikä rajapintaluokka ota kantaa metodin toteutuslohkon sisältöön. Käytettäessä rajapintoja varsinaisessa toteutusluokassa käytetään ilmaisua *implements*. (Harju & Juslin 2009, 211.)

Myös geneeriset luokat ovat toteutettavissa. Vesterholmin ja Kyppön (2003, 183-186) mukaan geneeriset luokat tarkoittavat parametrisoituja malleja joiden avulla voidaan para-

metrien tyypit kiinnittää muodostuviin samankaltaisiin luokkiin. Käytettäessä parametrisoitavaa luokkaa viitemuuttuja esitellään parametrin tyyppinä. Tämä ei ainoastaan rajoitu luokkiin vaan myös metodeissa voidaan käyttää parametrisyyttä.

Kieli tarjoaa ajoaikaisen ohjelman tarkastelun joita Ollikainen ym. (2010, 410-412) kutsuvat *reflektioksi*. Reflektio voidaan ymmärtää mekanismina, joka ohjelman ajoaikana voi tutkia ja muodostaa luokkia. Tyypillisesti reflektiossa käytetään staattisia metodeja *Class.forName()* ja *Class.newInstance()*. Ensimmäinen metodi toteuttaa luokan staattisen alustajan. Toinen metodi luo kokonaan uuden ilmentymän halutusta luokasta. Tällöin voidaan rajapinnan avulla kutsua ajoaikaisia luokkia, jotka toteuttavat kyseisen rajapinnan.

5.2 Paketinhallinta

Ohjelmistoprojektissa paketinhallinta toteutettiin Apache Mavenin avulla. Apache Maven on sovellusprojektin hallintatyöväline, joka rakentuu periaatteeseen *project object model, POM* (Apache Maven 2016). POM muodostaa Maven projektille yhden tiedoston nimeltä *pom.xml*, jossa on XML tietona projektin perustiedot, metatiedot, konfiguraatiot sekä riippuvuudet muihin sovelluksiin. Mavenin avulla voidaan rakentaa sovellus, testata, julkaista sekä rakentaa dokumentaatio projektiin. (Srirangan 2011. 19-27.)

Lisäksi Maveniin liittyy käsitteet *parent pom* ja *module project*. Ensimmäinen termi tarkoittaa Maven projekteilla perinnöllistämistä. Periävä projekti voi periä toisesta projektista konfiguraatioita, yhtenäisiä sovellusriippuvuuksia sekä ominaisuuksia ja muita resursseja. Toinen termi tarkoittaa modulaarista projektia, jossa projekti rakentuu eri projektin kokonaisuudesta. Modulaarisen projektin lapsiprojektit ilmoitetaan pääprojektissa. Tällöin projekti voidaan jakaa osiin, joissa yhdessä on tekniset toteutukset, toisessa palveluiden rajapinnat ja kolmannessa verkkoresursseja kuten kuvassa 11 on esimerkiksi kuvattu. (Lalou 2013, 19-28) Srirangan (2011, 28-31) lisää että rakentaessa sovellusta pääprojektista, Maven toteuttaa automaattisesti modulaarisen reaktori rakennuksen, jonka järjestyksen Mavenin sisäinen logiikka päättelee.

Kuva 11. Maven modulaariprojekti esimerkki rakenne (Lalou 2013, 19-28)

5.3 Ohjelmistonviitekehys Spring Boot

Williams (2014, 324) mukaan ohjelmistoviitekehys ei ole pakollinen osa ohjelmaa. Toisaalta, ohjelmistoviitekehys usein sisältää jo valmiiksi sellaisia ominaisuuksia, jotka tekevät kehittämisestä nopeampaa ja lähdekoodista testattavampaa.

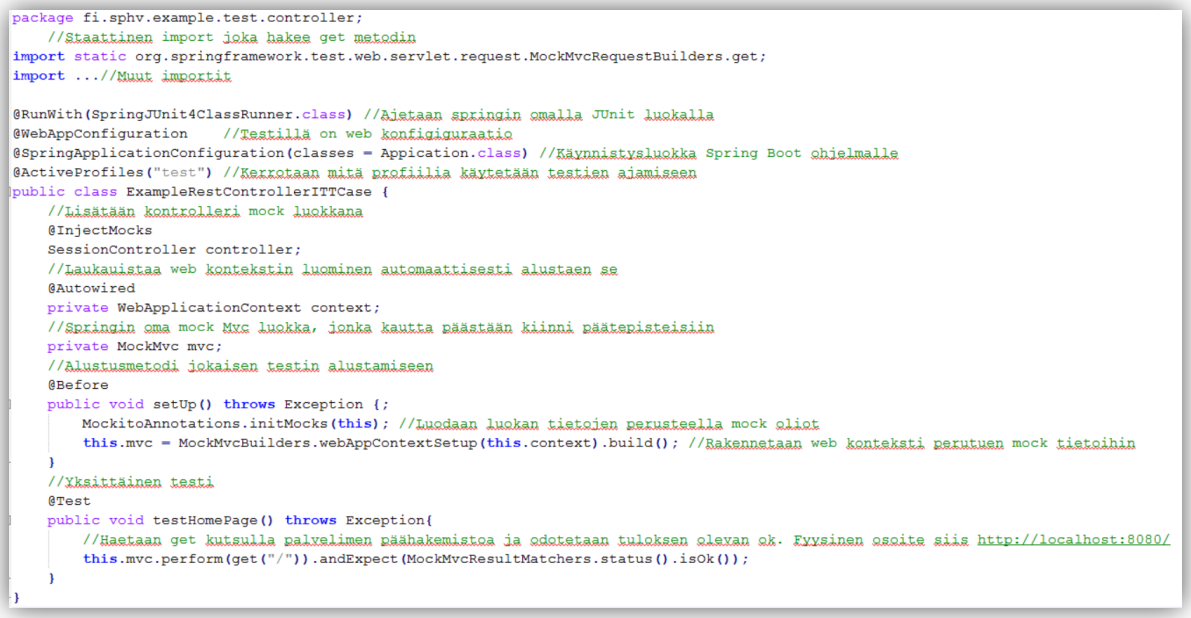
Tällainen ohjelmistoviitekehys on Spring Boot, jonka alkuperät ovat Spring-ohjelmistoviitekehyksessä. Sprint Boot on tosin irrallinen toteutus, joka ei erikseen vaadi toimiakseen erillistä XML-konfiguraatiota kuin alkuperäinen Spring. Spring Boot automaattisesti konfiguroi itsensä, mutta konfigurointia voidaan myös toteuttaa lisää tarpeen vaatiessa. Lisäksi sovellus rakentuu sisäänrakennetun virtuaalipalvelimen päälle, joten erillistä virtuaalipalvelimen asennusta ei tarvitse tehdä. Tuki löytyy muun muassa virtuaalipalvelimille Tomcat, Jetty ja Undertow. (Spring Boot 2016.) Lisäksi Spring Boot sisältää laajan aloituskirjastokokoelman, joita on mahdollista liittää projektiin. Aloituskirjastoita on muun muassa sovelluksen monitoroimiseen, tietokantayhteyksien muodostamisen sekä sivustojen sapluunamallit Thymeleaf ja Velocity malleille. Näitä voidaan esimerkiksi Mavenin kautta lisätä sovellukseen lisäämällä haluttu aloituskirjasto pom.xml tiedoston sovellus riippuvuuksiin. (Spring Boot Guide 2016a.)

Webohjelmointiin Spring Boot tarjoaa kirjastona *spring-boot-starter-web* paketin. Kirjasto tarjoaa muun muassa kontrollerirajapinnan, jonka avulla voidaan rakentaa MVC (model, controller, view) tapaisia toimintoja. Tällöin voidaan kirjoittaa web-applikaatioita, joita voidaan kutsua muun muassa selaimen http-kutsuilla. Sovellus lähettää käyttäjälle pyydetyn sivun tai vaihtoehtoisesti muun kutsun kautta tulevan sisällön. (Spring.Io 2016.)

Testauksessa viitekehys tarjoaa mahdollisuudeksi käyttää omaa kirjastoaan *spring-boot-starter-test.* Kirjasto sisältää integrointitestaukseen valmiita luokkia, jolloin kolmannen osapuolen kirjastoja ei välttämättä tarvitse käyttää. Kirjasto sisältää myös luokkia, joiden avulla testauksen alustuksessa käynnistetään ohjelma mikäli testaus sen tarvitsee. Lisäksi voidaan myös erikseen määrittää konfiguraatioita siitä, että missä portissa sovellus avataan. (Spring Boot Guide 2016b.)

Testauksessa voidaan myös testata erikseen kontrolleriluokkia, käyttämällä hyödyksi kirjaston *WebApplicationContext* sekä *MockMvc* luokkia. Luokkien avulla on mahdollista testitapauksen käynnistyessä luoda ilmentymä sovelluksesta, johon testitapaukset suoritetaan (Kainulainen 2013). Tämä on esitetty kuvan 12 esimerkissä. Kuvan luokassa suoritetaan *SessionController*-luokalle integrointitesti, jossa lähetetään http-kutsu virtuaalipalvelimen juureen osoitteeseen *http:///localhost:8080/* ja odotetaan vastausta kyseiselle kutsul-

le. Vastauksen ollessa toivottu, testi menee lävitse. Kirjasto kääntää koko lähdekoodin toimivaksi ohjelmaksi, jolloin testi suoritetaan samankaltaisesti kuin manuaalitestaus. Tällöin voidaan suorittaa integrointitestaus ilman erillisiä kolmannen osapuolen kirjastoja ja käytetään Springin omia ominaisuuksia hyödyksi.



```
package fi.sphv.example.test.controller;
    //Staattinen import joka hakee get metodin
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import ...//Muut importit

@RunWith(SpringJUnit4ClassRunner.class) //Ajetaan springin omalla JUnit luokalla
@WebAppConfiguration    //Testillä on web konfigiguraatio
@SpringApplicationConfiguration(classes = Appication.class) //Käynnistysluokka Spring Boot ohjelmalle
@ActiveProfiles("test") //Kerrotaan mitä profiilia käytetään testien ajamiseen
public class ExampleRestControllerITTCase {
    //Lisätään kontrolleri mock luokkana
    @InjectMocks
    SessionController controller;
    //Laukaistaa web kontekstin luominen automaattisesti alustaen se
    @Autowired
    private WebApplicationContext context;
    //Springin oma mock Mvc luokka, jonka kautta päästään kiinni päätepisteisiin
    private MockMvc mvc;
    //Alustusmetodi jokaisen testin alustamiseen
    @Before
    public void setUp() throws Exception {;
        MockitoAnnotations.initMocks(this); //Luodaan luokan tietojen perusteella mock oliot
        this.mvc = MockMvcBuilders.webAppContextSetup(this.context).build(); //Rakennetaan web konteksti perutuen mock tietoihin
    }
    //Yksittäinen testi
    @Test
    public void testHomePage() throws Exception{
        //Haetaan get kutsulla palvelimen päähakemistoa ja odotetaan tuloksen olevan ok. Fyysinen osoite siis http://localhost:8080/
        this.mvc.perform(get("/")).andExpect(MockMvcResultMatchers.status().isOk());
    }
}
```

Kuva 12. Esimerkkitestitapaus kontrollerin testaukseen

5.4 Jatkuva integrointi, toimitus ja käyttöönotto

Jatkuva integrointi *(continuous integration)* on Leffingwellin mukaan (2008, 169-172) prosessi, jossa rakennustyövälineillä automatisoidaan lähdekoodin muutoshallintaa. Hallinta muodostuu keskitetystä lähdekoodin säilytyspaikasta. Täältä kehittäjät voivat ottaa käyttöönsä lähdekoodin sekä tallettaa uutta lähdekoodia versionhallintaan. Päivittäin tämä versionhallinta käännetään rakennusautomaatin avulla, josta tutkitaan kääntyykö lähdekoodi ohjelmaksi. Tavanomaista on myös se, että lähdekoodi sisältää yksikkötestejä sekä mahdollisia integraatiotestejä, jotka ajetaan käännöksen ohessa. Tuloksesta kääntäjä kerää palautteen ja siitä kehittäjät voivat tarkastella ja arvioida syntynyttä tulosta.

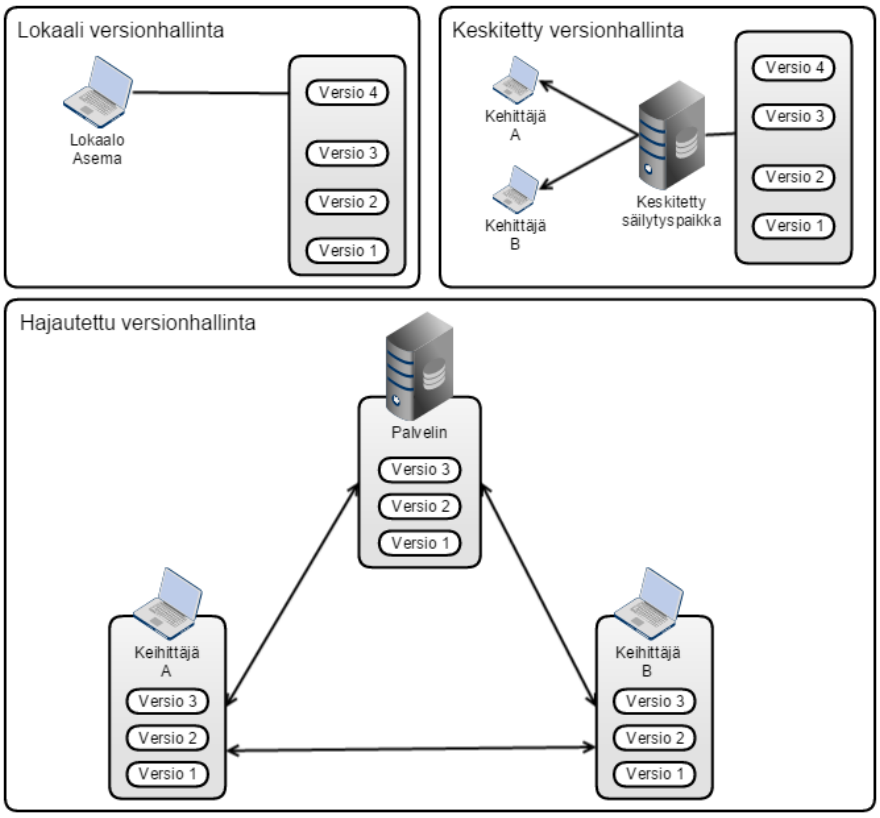
Fowler (2013) puolestaan kuvailee että jatkuva toimittaminen (*continuous delivery*) on puolestaan prosessi, jossa sovellus voidaan julkaista tuotantoon koska tahansa. Tätä ei kuitenkaan ole pakonomaista tehdä. Jatkuva toimittaminen vaatii jatkuvan integrointivaiheen, jossa suoritetaan testaus lähdekoodiin tuotantoa vastaavassa ympäristössä. Erona jatkuvaan käyttöönottoon (*continuous deployment*) on se, että käyttöönotossa koko julkaisuketju on automatisoitu. Pääasiallisesti ketju näyttää kuvan 13 mukaiselta ketjulta. Käy-

tännössä erona jatkuvalla käyttöönotolla ja toimittamisessa on viimeisen vaiheen suorittaminen.

Kuva 13. Toimitusketjun malli

5.5 Versionhallinta

Somasundaramin (2013, 8-9) mukaan järjestelmää, joka pystyy tallentamaan tiedoston tai tiedostojen muokkaukset tietyltä ajanjaksolta ja kykenee palauttamaan vanhemman version tiedostosta tai tiedostoja, voidaan kutsua versionhallintajärjestelmäksi. Lisäksi tiedostoja voidaan merkitä erilaisilla avainsanoilla. Järjestelmä havaitsee muutokset tiedostoissa sekä pitää historiatiedon koko ajasta, kun versionhallinta on otettu käyttöön. Tyypillistä versionhallinnalle on, että se rakentuu eri vaiheisiin. Kuvassa 14 on kuvattuna erilaisia versionhallintatapoja.



Kuva 14. Versionhallintatapoja (Git 2016)

Somasundaramin (2013, 11-15) mukaan versionhallinta voi olla lokaalia, keskitettyä, tai hajautettuna, jotka ovat esitetty kuvassa 14. Lokaalissa versionhallinnassa tiedot sijaitsevat yksinkertaisessa tietokannassa. Tosin tieto on kiinni yhdessä koneessa. Keskitetyssä versionhallinnassa varsinainen versionhallinta sijaitsee palvelimella, jota kehittäjät käyttävät eri koneilta. Ongelmana tässä tavassa on palvelin mikäli se kaatuu tai palvelimeen tehdään muutoksia, jotka estävät kehittäjiä hakemasta tietoja palvelimelta. Historiatiedotkin sijaitsevat palvelimella eikä kehittäjien koneella. Hajautetussa versionhallinnassa ei ainoastaan haeta viimeisintä tietoa vaan koko historia peilataan käyttäjän koneelle. Tällöin palvelimen kaatuessa historiatiedot sijaitsevat jokaisen kehittäjän koneella.(Git 2016.)

5.6 Lähdekoodin laadunhallinta

Jotta varmistutaan, että laadukkuus rakentuu ohjelmiston sisälle, tulee sen olla analysoitavissa. Swartoutin (2012, 107-111) mukaan analysoitavia tietoja ovat muun muassa lähdekoodin kommentoiminen, monimutkaisuus, testikattavuus, versionhallintaan tehtyjen muutosten määrä, käyttämätön ja monistetun lähdekoodin määrä.

Laadunhallinnan eräs työväline on SonarQube-niminen järjestelmä. Ohjelma analysoi lähdekoodia monesta eri näkökulmasta ja luo siitä joukon erilaisia raportteja. Analysointi ei ainoastaan rajoitu lähdekoodiin, vaan myös ohjelmiston rakenteisiin, dokumentointiin, tekniseen haastavuuteen sekä testikattavuuden analysoimiseen. Tällöin saadaan metriikkaa useasta osasta ohjelmaa sekä tieto pystytään muuttamaan arvoiksi, jotka tuottavat laadukkuutta tuotteelle. Ohjelma tallettaa myös ongelmat ja sekä niihin liittyvät tiedot. Tämän lisäksi ohjelma säilyttää historiatietoa, jota voidaan verrata ohjelmiston eri kehitysvaiheiden välillä.(Charalampos 2012. 7-11.)

5.7 Komentorivin komentosarjat

Lakshman (2011, 8-10) toteaa Komentorivin kautta voidaan ajaa komentosarjoja eli skriptejä, joilla voidaan hallita käyttöjärjestelmää, joista yleisin on GNU/Linux käyttöjärjestelmissä käytössä on Bash (*Bourne Again Shell*) komentorivi ympäristö. Blum (2008, 13-14) jatkaa, että komentorivin kautta voidaan suorittaa joukko erilaisia tehtäviä. Esimerkiksi voidaan kopioida, siirrellä ja uudelleen nimetä tiedostoja, avata ja sulkea ohjelmia sekä prosesseja sekä suorittaa komentosarjoja. Myös käyttöjärjestelmän tietoja voidaan tutkia. Lisäksi komentoja voidaan suorittaa tietyillä ohjelma sanoilla, jotka komentorivitulkki ymmärtää kääntää koneelle suoritettaviksi tehtäviksi.

Blum (2008, 202) kertoo, että skriptit ovat käyttäjän määrittelemiä omia suoritettavia tehtäviä. Skriptit alkavat ensimmäisellä rivillä *!#/bin/bash* jonka jälkeen tulee muut komennot.

Ensimmäisen rivin tarkoituksena on kertoa komentorivi ympäristölle, missä ympäristössä komentosarja tulisi suorittaa. Komennot suoritetaan esiintymisjärjestyksessä. Jotta komentosarja on suoritettava, tulee se muuttaa ajettavaksi komennolla *chmod u+x*.

Blummin (2011, 749-757) mukaan bash sisältää suuren joukon valmiita komentoja. Komennot jaetaan sisäänrakennettuihin ja ulkoisiin komentoihin. Komentoja voidaan ketjuttaa omiin skripteihin ja komentojen avulla voidaan suorittaa melkein mikä tahansa komentorivin kautta suoritettava toiminto. Taulukossa 1 on kuvattuna yleisimpiä komentoja, joita voidaan suorittaa komentorivin kautta.

Taulukko 1. Bash komentoja

Kommenon lyhenne	Tarkoittaa
history	Näyttää viimeiseksi suoritettuja komentoja
kill	Lähettää viestin, jonka avulla voidaan lopettaa prosessi sen id arvolla (PID)
pwd	Näyttää tämänhetkisen työskentely hakemiston
chmod	Muuttaa tiedoston tai hakemiston oikeuksia
chown	Muutta tiedoston tai hakemiston omistajuutta
cp	Kopiointi komento
ls	Listaa hakemiston tiedot
mkdir	Luo hakemiston
mv	Tiedoston siirto ja uudelleen nimäminen
rm	Tiedoston poistokomento
zip	Paketoi tiedoston zip muotoon

Blumm (2011, 210-213) toteaa myös, että komentorivi ohjelman avulla voidaan myös kirjoittaa suoraan tiedostoon. Tämä tapahtuu kirjoittamalla komennon jälkeen suurempi kuin merkki, jonka jälkeen annetaan tiedoston nimi. Pienempi kuin merkillä luetaan tiedoston sisältö komentoon. Mikäli käytetään tuplasti suurempi kuin merkkiä, tarkoitus muuttuu koko tiedoston kirjoittamisesta ainoastaan uuden tiedon lisäämisestä tiedostoon. Tupla pienempi kuin merkkiä puolestaan aiheuttaa sen, että kahden merkin välissä oleva sisältö suoritetaan.

6 Automatisoitavan elinkaaritestauksen kehitystyö

Kehitystyö toteutettiin ketteränä ohjelmistokehitysprojektina neljässä iterointikierroksessa. Ohjelmistokehitystyö toteutettiin toimeksiantajan tiloissa ja laitteilla, jolloin käytännön asioiden hoitaminen oli helpompaa, kuten myös salassapitovelvollisuuden hallinnointi. Kehitystyövälineet sijaitsivat toimeksiantajan laitteissa, jolloin lähdekoodin hallinta pystyttiin kapseloimaan toimeksiantajan omiin järjestelmiin. Keskeisimpinä hallinnointityövälineinä käytettiin toimeksiantajan Atlassian-tuoteperhettä, Sonatype Nexusta ja SonarQubea.

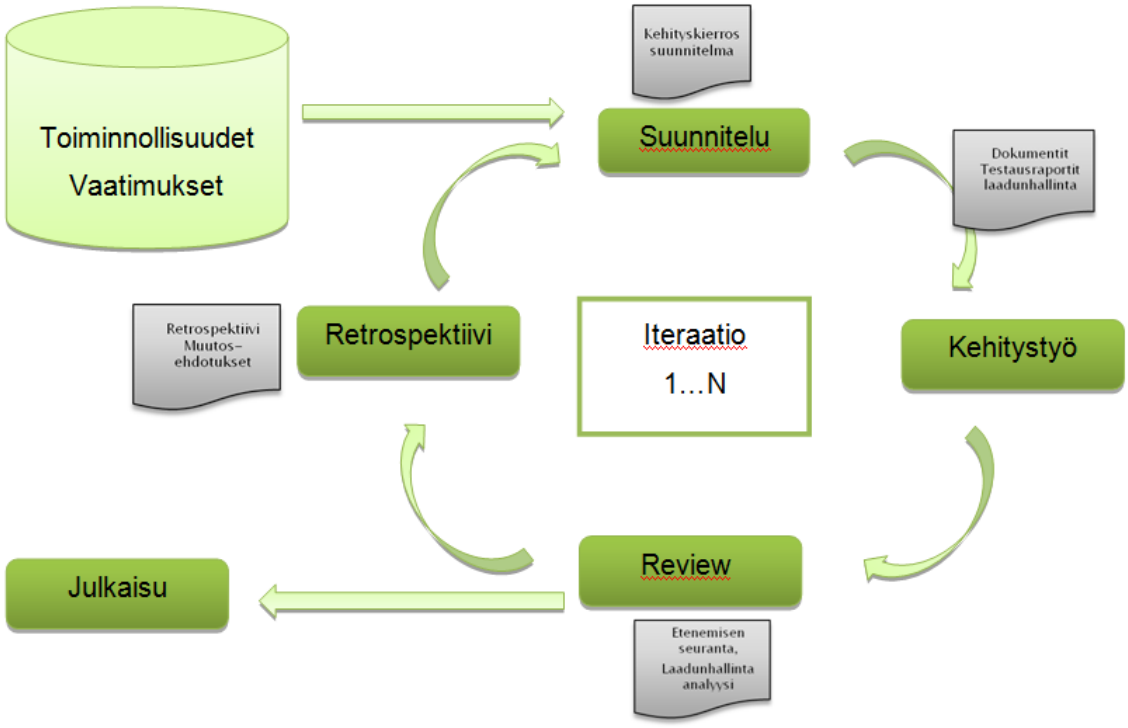
Automatisoinnilla pyrittiin ratkaisemaan toimeksiannossa sellaisten töiden tekeminen, jotka muuten veisivät manuaalisesti enemmän aikaa toteuttaa. Tämä automatisointi pyrki toteuttamaan automatisoinnin niin laadukkaasti, että tuotettu viiteaineisto olisi virheetöntä. Tällä saavutettaisiin viiteaineistolle asetetut laatutekijät sekä viiteaineisto olisi määritysten mukainen. Tuotetut viiteaineistot ovat laadultaan sellaisia, jotka testattava tietojärjestelmä pystyy käsittelemään onnistuneesti.

Projektinhallinnollisissa menetelmissä kevennetty scrum mahdollisti yhtenäisen projektinhallinnollisten menetelmien hallinnoimisen sekä standardoi käsitteitä, jolloin yhtenäinen käsitteistö selvensi tekemistä. Scrumia ei kuitenkaan täysin voitu soveltaa kehitysprosessina, koska scrum ei salli iterointikierroksen aikana muutosta, joka muuttaisi kehityskierroksen keskeistä tavoitetta. Tällöin olisi voinut syntyä tilanne, jossa kehityskierros voisi jäädä tilaan, jossa edistymistä ei tapahtuisi huonosti suunnitellun kehitysjonon vuoksi. Lisäksi kehitystyö sai osakseen todella paljon kokeellista kehittämistä, jossa prototyyppistä kehittämistä syntyy päivittäin, mikä ei aina mahdollista yhden tuoteominaisuuden valmistumista yhden kehityspäivän aikana. Yhden henkilön ohjelmistokehitysprojektissa scrum ei soveltunut yksinään käytettäväksi kehitysmenetelmäksi, jolloin pääasiallinen hyöty saatiin projektinhallinnallisista menetelmistä. Yhteinen kieli, ymmärrys mitä tehdään ja miten, auttoivat selventämään päivittäistä työskentelyä. Tällöin yhdistämällä ketterän kehittämisen, leanin sekä kanbanin tapoja ja ajatuksia rakennettiin oma kehitysprosessi, jossa kehitystiimin muodostuessa yhdessä henkilöstä pystyi toimimaan omatoimisesti ja tehokkaasti. Tuoteomistajan roolissa toimi toimeksiantajan edustaja, jonka pääasiallisena tehtävänä oli tuottaa tuotettavalle ohjelmistoratkaisulle tuotteen kehitysjono.

Lean ajattelun tavalla projektissa pyrittiin minimoivaan turhan työn tekeminen, ja pyrittiin tuottamaan ohjelmistoratkaisulle arvoa ensimmäisestä kehityskierroksesta alkaen. Hukkien minimoimiseksi suunniteltiin kehityskierroksen aikana mitä tullaan kehittämään. Jotta varmistettaisiin laadukkuuden rakentuminen tuotteen osaksi, työssä käytettiin jatkuvaa integrointia, versionhallintaa sekä SonarQube analysointia. Tietämystä projektin tilantees-

ta ja toiminnollisuuksista pidettiin yllä toimeksiantajan omien järjestelmien avulla, jotka koostuivat Atlassian tuoteperheestä. Yhteisillä työvälineillä mahdollistettiin läpinäkyvyys, jolloin toimeksiantajan organisaatiolla oli koko kehitysprosessin ajan mahdollisuus monitoroida projektin etenemistä. Kokonaisuuden optimointia suoritettiin näyttötilaisuuksissa, jotka olivat kehityskierroksen viimeisenä päivänä. Tehdyt ominaisuudet käytiin yksitellen lävitse ja parannusehdotukset kirjattiin ylös.

Kehitystyö toteutettiin inkrementaalisesti kuvan 15 mukaan neljässä kehityskierroksessa. Jokainen iterointikierros aloitettiin suunnitteluvaiheella, jossa valittiin toimeksiantajan edustajan kanssa toimintoja kehitykseen. Kehitykseen valitut osa-alueet jaettiin vielä erikseen pienempiin osa-alueisiin, jotta nähtäisiin paremmin, mistä osakokonaisuuksista toiminto koostui. Samalla huomattiin määrittämättömiä toimintoja, jotka piti tuottaa toiminnon valmistuksen ohessa. Esimerkiksi IBAN-muotoisen tilinumeron generointi oli toiminnollisuus, joka toteutettiin toisen toiminnollisuuden osana. Kehitystyössä valittuja toiminnollisuuksia kehitettiin siten, että toiminnollisuuteen liitettiin dokumentointi, testaus sekä muut mahdolliset lisätiedot. Kierroksen lopuksi pidettiin näyttötilaisuus (*review*), jossa iterointikierroksen tulos esiteltiin toimeksiantajan edustajalle. Näyttötilaisuudessa saatiin toimeksiantajalta arvokasta palautetta, jota käytettiin hyväksi seuraavan kierroksen suunnittelussa. Kierros johti julkaisun lisäksi retrospektiiviin, jossa kehitystiimi mietti mitä tehtiin hyvin ja mitä olisi voitu tehdä toisin.



Kuva 15. Kehityskierros yksinkertaisuudessaan

Kehitystyön suurimmaksi hyödyksi osoittautui heti Atlassian tuoteperhe, joka sulavasti pystyttiin liittämään jokaiseen projektin piirteeseen. Confluence mahdollisti keskitetyn projektin dokumentin hallinnan. JIRA toteutti tehtävienhallinnan sekä projektin raportointijärjestelmän. Versionhallinta toteutettiin BitBucket, entiseltä nimeltä Stash, järjestelmällä, joka toimii Git versionhallinnan avulla. Tällöin pystyttiin luomaan projektin lähdekoodin säilytyspaikka ja monipuolinen jaettavuus hajautetun versionhallinnan kautta. Varsinaisena rakentaja automaattina, joka toteutti jatkuvan integroinnin ketjun, valittiin Bamboo niminen järjestelmä. Tuoteperheen ulkopuolelta Sonatype Nexus tarjosi rakennetuille paketeille säilytyspaikan ja vaivattoman lataamisen muihin projekteihin sisäverkosta. Viimeisenä pääasiallisena työvälineenä toimi SonarQube, joka on lähdekoodin laadunhallinta työväline.

Dokumentinhallinta toteutettiin Confluencen avulla, jossa jokainen dokumentti pystyttiin tallentamaan. Järjestelmä mahdollisti myös dokumentaation luomista erillisinä sivuina, jonne pystyttiin makrojen avulla tallentamaan lisätietoja tarvittaessa. Toissijaisena dokumentin tuottotapana käytettiin JavaDoc tuotoksia sekä SpringFoxia, jonka avulla pystyttiin dokumentoimaan jokainen kontrolleri, joita pystyttiin käyttämään Curl-kutsujen kautta.

Tehtävienhallintajärjestelmän käytettiin Atlassian JIRA nimistä järjestelmää. Järjestelmään merkittiin toimeksiantajan puolesta priorisoituna kaikki tarinat, jotka jaettiin pienempiin tehtäviin iterointikierroksen alussa. Kierroksien alussa siirrettiin sopiva määrä toteutettavia tehtäviä kierrokselle. Jokaiseen tehtävään pystyttiin vaivatta liittämään lisätietoja mihin komponenttiin tehtävä liittyi ja mikä sen tämän hetkinen tila oli. JIRA mahdollisti myös kätevästi projektin tilan katsomisen erillisestä muokattavasta hallintopaneelista ja tarjosi laajan valikoiman erilaisia raportteja projektinhallinnan tueksi.

Versionhallintajärjestelmänä käytettiin Atlassian BitBucket järjestelmää. Järjestelmä toimii Git versionhallinnan avulla. Järjestelmä tarjosi Web käyttöliittymän kautta uusien säilytyspaikkojen luomisen sekä lähdekoodin haaroittamisen. Haaroittamisen pystyi myös suorittamaan JIRA:n kautta luomalla järjestelmälinkkejä järjestelmien välille. Tällöin yhdessä haarassa pystyttiin tuottamaan yhden toiminnollisuuden tuottaminen. Kun toiminnollisuus oli valmis ja testattu, oli vaiheena liittää toiminnollisuushaara kehityshaaraan. Jokaisen iteraation loppuvaiheessa, kehityshaara uusine toiminnollisuuksiin oli mahdollista liittää tuotantohaaraan.

Bamboo on Jenkinsin kaltainen jatkuvan integroinnin työväline. Järjestelmän avulla voidaan rakentaa ja testata päivittäin useita kertoja lähdekoodia, joka on tallennettuna BitBucket lähdekoodin säilytyspaikasta. Bamboossa luodaan aluksi projekti, jossa määritel-

lään projektille nimi ja määritetään lähdekoodin säilytyspaikka. Tämän jälkeen määritetään erillinen työ, jossa lähdekoodi haetaan. Tämän jälkeen voidaan esimerkiksi Mavenin avulla kääntää ohjelma ja ajaa lähdekoodista löytyvät testit. Mikäli testit menevät läpi, ohjelma kääntyy ja Bamboo ilmoittaa ohjelman suorituksen tilan.

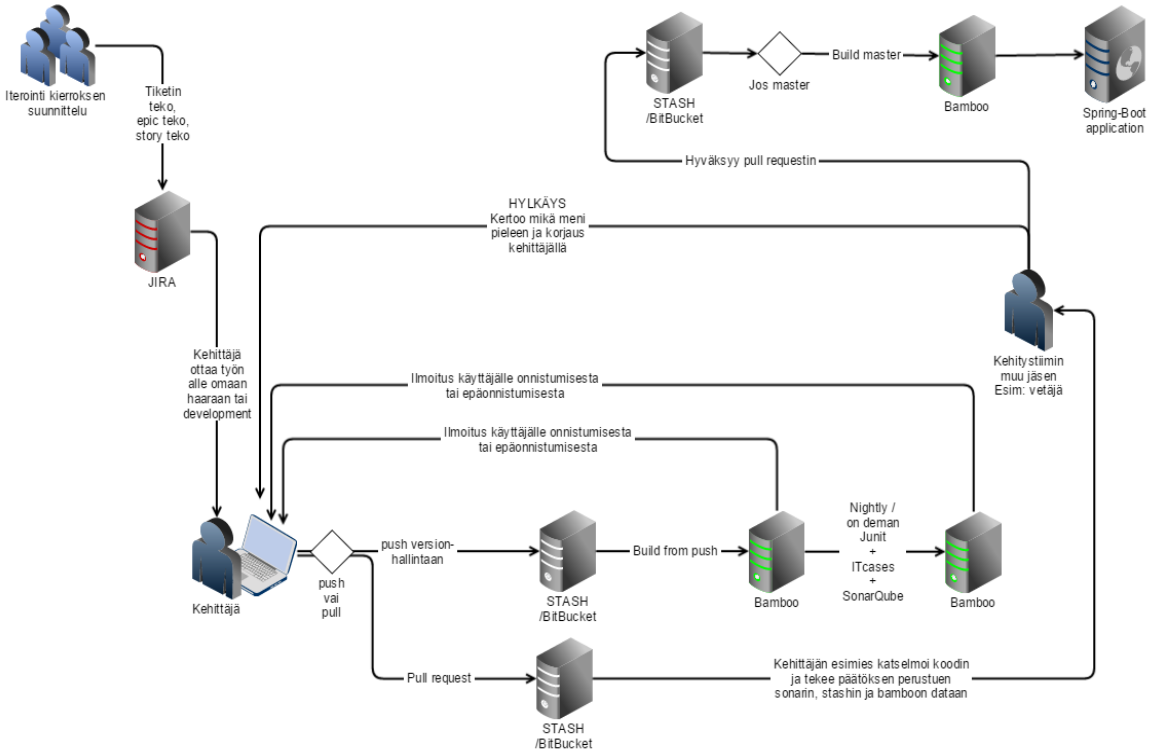
Sisäisenä paketinhallintajärjestelmänä toimi Sonatype Nexus, jonne tallennettiin lähdekoodin käännetyt paketit. Paketit pystyttiin näin siirtämään pois käyttäjän lokaalilta asemalta kaikkien kehittäjien saataville. Lataus pystytään tekemään muun muassa Mavenin sovellusriippuvuuksien avulla. Tällöin tosin tulee erikseen määrittää *pom.xml* Nexuksen sijainti.

Lähdekoodin laatua niin testikattavuuden, rakenteen sekä dokumentoinnin kautta, analysointiin ja hallinnointiin SonarQuben avulla. Analysointi toteutettiin Bamboon clover lisäosan avulla, jossa rakennuksen aikana cloverin data lähetetään SonarQubeen. Tämän datan SonarQube kääntää luettavaan muotoon, ja piirtää sen pohjalta mitattavia tietoja. Tietoja käytettiin löytämään vikoja, puutteita ja oman oppimisen tukena, jolloin käytännössä näki ne osa-alueet ohjelmistosta, jossa oli kehitettävää.

Varsinainen työ toteutettiin aluksi useassa Maven projektissa, joka toisessa iterointikierroksessa muutettiin Maven modulaariprojektiksi. Useassa osassa toteuttamisen ideana oli luoda erillinen Spring Boot projekti, jossa testiohjelman viitekehys toimisi ajon aloittavana ohjelmana. Ohjelmaan liittyi kaksi projektia, jossa ensimmäisessä sijaitsi kaikki rajapinnat sekä abstraktit luokat ja yleisiä yläluokkia. Toisessa projektissa oli käyttökelpoisia apuohjelmia, jotka tuottivat yleishyödyllisiä muuntimia ja rajoittimia, joita pystyttiin käyttämään projektin ulkopuolisissa projekteissa. Neljäs projekti, joka liitetään rakennusvaiheessa koko ohjelmaan, on varsinainen testattavaan ohjelmistoon liittyvä projekti, joka toteuttaa rajapintaprojektin ja mahdollisesti käyttää hyväkseen yleisprojektia. Tähän projektiin määritetty testattavalle ohjelmistolle luodut luokat, jotka ovat sille ominaisia. Tällöin jatkossa voidaan rakentaa mahdollisesti uusia projekteja, jotka testaavat eri ohjelmaa tai kilpaileva uusi projekti edellisen tilalle.

Päivittäinen työrytmi on kuvattuna kuvaan 16. Kehittäjän roolissa JIRA:n kautta hallittiin iterointikierroksen työmäärää. Työn otettaessa kehitykseen, kehittäjä pystyi kätevästi luomaan version kehityshaaran, jossa työ toteutettiin. Tällöin kuvassa mennään *push* reittiä pitkin. Kun uutta lähdekoodia oli lisätty versionhallintaan palvelimelle, huomasi jatkuvan integroinnin palvelu tämän ja aloitti välittömästä lähdekoodin kääntämisen. Lähdekoodin mukana siirtyi testitapaukset, jotka ajettiin kääntämisen yhteydessä. Nämä tosin olivat yksikkötestejä. Integrointitestit sekä yksikkötestit ajettiin öisin ja halutessa. Samalla suori-

tettiin lähdekoodin analysointi, jonka tulos siirrettiin SonarQubeen. SonarQube suoritti laajan analysoinnin ja aamuisin kehittäjillä oli katselmoitavissa sekä Bamboon tulos sekä SonarQuben luomat analyysit. Näiden avulla löydettiin kehittämiskohteet ja kriittisimmät virheet nopeasti.



Kuva 16. Kehitystyön eteneminen

Mikäli kyseessä olikin *pull* tyyppinen toiminto, tarkoitti tämä kehitysversiohaaran yhdistämistä toiseen haaraan. Useimmissa tapauksissa yhdistäminen koski kehityksen päähaaraan toiminnollisuuden liittämistä. Kun *pull* kutsu lähetettiin, ajettiin samalla testit. Mikäli testit menivät lävitse, voitiin kehityshaara yhdistää turvallisesti toiseen haaraan. Jos tämä yhdistämishaara oli tuotantohaara, reitti muuttui siten, että Bamboossa ajettiin yksi ylimääräinen rakennuskerta, josta syntyi tuotantoon asennettavissa oleva paketti.

Ohjelmistoratkaisussa rakennettiin suoritettavat työt omina luokkina, jotka toteuttivat abstraktin yliluokan. Työt olivat liiketoiminnan kautta luotuja toiminnollisuuksia, jolloin jokaista työtä vastasi manuaalisesti toteuttavat sama tehtävä. Töihin liitettiin työn nimike, konfiguraatio sekä kaikki työn työnvaiheet. Jokaiseen työvaiheeseen voidaan liittää myös oma konfiguraatio sekä parametristö, jonka mukaan työvaihe toimii. Työvaiheeseen liittyy myös järjestys, jossa järjestys toimii kokonaislukujen avulla pienimmästä suurimpaan. Tämä mahdollisti töiden ketjuttamisen, jolloin yhteen työhön voitiin liittää useita eri vaiheita.

Tuotettu ohjelma sisälsi paljon rajapintojen sekä yliluokkien käyttämistä. Tämä mahdollisti sen, että ohjelmistossa pystyttiin yliluokkien ja rajapintojen kautta lähestymään ilmentymiä tuntematta tarkalleen mikä se on. Luokan toteuttaessa rajanpinnan, voitiin parametrisoidut komennot ajaa suoritettavassa ohjelmassa kyseisten parametrien mukaisesti. Tämä mahdollistaa dynaamisen toteuttamisen, ja mahdollisen jatkokehittämisen vaivattomuuden ettei kaikkia luokkia tarvitse uudelleen muuttaa. Rajapintaluokkia pystyttiin myös lähestymään reflektion kautta, jolloin tieto pystyi liikkumaan vaivattomasti. Esimerkkinä rajapintaluokasta on ohjelmistoratkaisussa lokien kirjoittaja, jonka rajapintaa käytetään hyväksi muissa luokissa, jolloin lokien varsinainen kirjoittajaluokka esitellään vain kerran työn alkaessa. Näin ei tarvitse jokaisessa luokassa alustaa uutta muuttujaa kirjoittamaan lokeja.

Ohjelmointikielen avulla toteutettiin myös geneerisiä luokkia, joiden avulla tietojen käsitteleminen oli helppoa. Geneeriikan avulla pystyttiin ilmentymien tietoja käsittelemään ja tallentamaan väliaikaisiin tiloihin, joista tieto pystyttiin hakemaan. Tietokantakyselyissä geneerinen ratkaisu oli käytännöllisin. Tämän kaltainen ratkaisu poisti yksittäisten Java luokkien muodostamisen jokaista tietokantaobjektia kohden, jolloin geneerisessä luokassa tallennettiin tietokannan sarakkeen nimi ja arvo. Myöhemmin tämä arvo pystyttiin käyttämään hyväksi parametrisuuden avulla, jossa kerrottiin mikä sarakkeen arvo tulee mihinkin kohtaan tulosteessa. Toteutuksessa on myös hyvänä puolena se, että ohjelmallista lähdekoodia ei tarvitse muuttaa, mikäli tietokannan taulun sarakkeiden nimet muuttuvat vaan muutos toteutetaan työn parametristoon.

Tärkein ominaisuus koko työn kannalta oli satunnaisgeneraattorin luominen. Liitteessä 4 on avattuna satunnaisgeneraattorin toimintaidea. Satunnaisgeneraattorin toiminta perustuu neljään vaiheeseen. Mikäli satunnaistapahtumia tapahtuu, suoritetaan tapahtumien lataus. Lataustapahtumassa ladataan vain ne tapahtumat, jos niiden ennakkoehdot voivat täyttyä. Toisena tapahtumana arvotaan, mitkä tapahtuvat tulevat tapahtumaan. Kolmas vaihe sisältää tapahtumien määrän, eli kuinka monta kertaa tapahtuma tapahtuu.Neljäntenä vaiheena suoritetaan tapahtumien generointi ja niiden kirjaaminen päiväkirjaan.

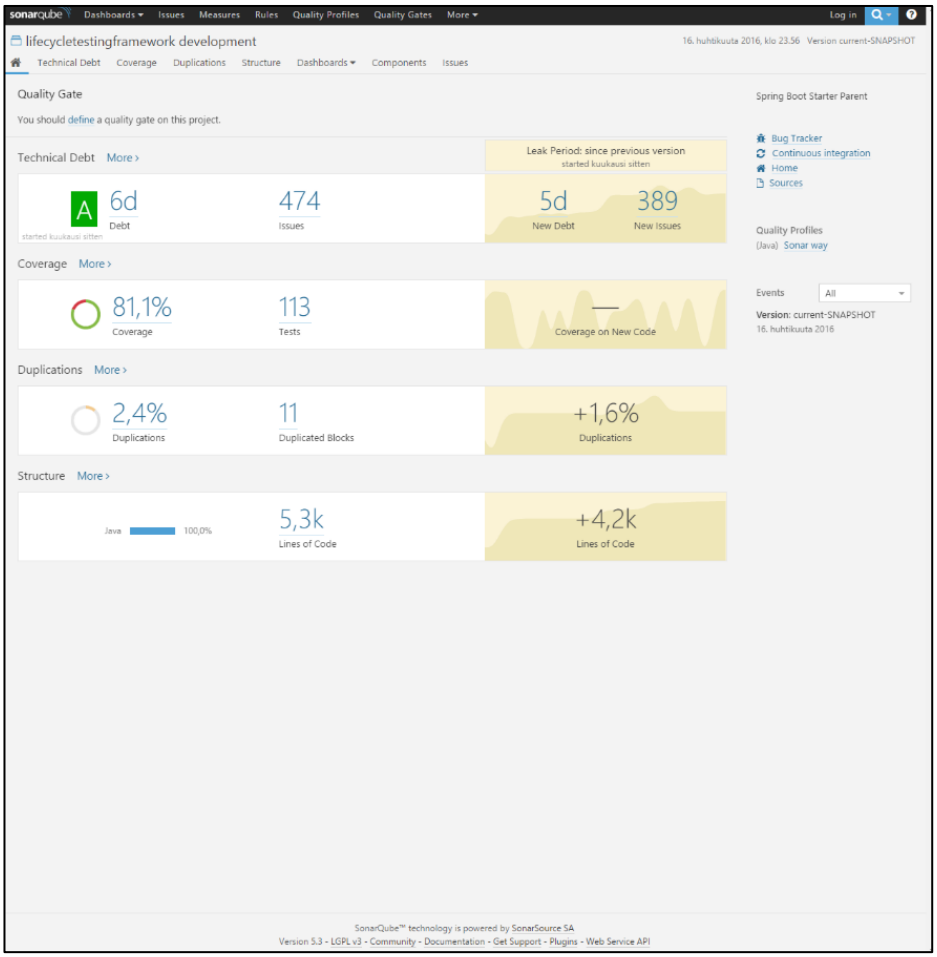
Varsinainen automatisoitu elättäminen tapahtuu komentosarjaketjun avulla, jonka toiminnan idea on kuvattuna liitteessä 5. Komentosarja toimii kolmen parametrin myötä: ohjelmiston kotihakemisto, parametrisoitujen työkonfiguraatioiden hakemisto sekä kokonaisluvun avulla. Hakemistojen avulla komentosarja automaattisesti käynnistää viiteaineistojen generoinnin sekä ohjelman sammuttamisen. Kokonaisluku kertoo, kuinka monta päivää elätetään järjestelmää eteenpäin. Kokonaisuudessaan komentosarja on pitkä ja sisältää paljon hakemistojen välillä liikkumista, tiedostojen siirtämistä, kopioimista, siivoamistöitä ja

kellon siirtoja. Myös analysointi suoritetaan komentosarjan kautta, jossa käynnistymis- ja sulkemisajat otetaan ylös. Näiden lukujen erotuksesta saadaan aika, kuinka kauan komentosarjakohtien välillä kului aikaa. Komentosarjassa on myös tutkiva vaiheita, jotka tutkivat nykyistä päivämäärää. Esimerkiksi viikonpäivien erottaminen viikonlopuista on eräs päivämääriin liittyvä tutkimuskohde. Viikonloppuisin ei ole tarkoituksenmukaista tuottaa viiteaineistoja vaan ainoastaan arkipäivinä.

7 Tuloksien analysointi

Työn lopputulos oli toimeksiantajan mieleinen. Työ koostui neljästä kehityskierroksesta, joiden aikana jokaiseen kehityskierrokseen muodostui eräänlaisia teemoja, joiden pohjalta kehittäminen eteni. Ensimmäisessä kehityskierroksessa teemana oli yleisilmeen ja ohjelmiston viitekehyksen rakentaminen. Toisessa kierroksessa keskityttiin aineistojen luomiseen aineistojen vaatimusmääritysten mukaisesti. Mallikappaleet vaatimuksineen ja ohjeineen toimitettiin toimeksiantajan kautta. Kolmannessa kierroksessa rakennettiin satunnaistapahtumille simulaattori, jolloin viimeisessä kehitysvaiheessa jäljelle jäi elätyskierroksien ajaminen.

Jokaisessa kehityskierroksessa arviointiin SonarQube-ohjelmiston avulla tuotetun ohjelmiston laadukkuutta. Laadun tutkiminen perustui ohjelman sisäiseen laadukkuuteen, jota mitattiin teknisen velan, testikattavuuden, toiston ja rakenteen avulla. Kuvassa 17 on esitetty mitatut laadut. Ne ovat viimeisen kehityskierroksen tulos ennen päähaaraan yhdistämistä.



Kuva 17. SonarQuben analysointi viimeisestä kehityskierroksesta

Tuloksesta on hyvä huomioida, että tekninen velka on SonarQuben arvion mukaan kuuden päivän mittainen, joka muodostuu 474 ongelman myötä. Testikattavuus on 81,1% ja se on saavutettu 113 testin kautta. Testit sisältävät sekä komponentti että integrointitestausta. Toistoa esiintyy 2,4% verran ja koskee 11 tapausta. Yhteensä ohjelmistokoodia on noin 5,3 tuhatta riviä.

Arvojen perusteella työssä on kertynyt teknistä velkaa, jonka SonarQube on itse arvioinut omien sääntöjensä avulla. SonarQube myös näyttää, millaisesta velasta on kyse ja useimmiten korjauksen ongelmanratkaisuun. Kun testikattavuus on yli 80% voidaan todeta, että suurin osa lähdekoodista sisältyy testeihin, mutta ei kuitenkaan aivan kaikki. Toistuvat rakenteet saattavat selittyä osin ohjelmointikielen kautta, jossa metodien sisällä esiintyy identtisiä osia, jotka voisivat olla erillinen luokka. Rivien määrän perusteella voidaan laskea, että testeihin sisältyy noin 4,3 tuhatta riviä, jolloin noin tuhat riviä ei sisälly testeihin. Tämän kaltaisia rivejä voivat olla virheenheittorivit, joita ei kaikkia ei käydä lävitse testeissä.

Jo ensimmäisen kehityskierroksen aikana syntyi toimeksiantajan toivomusten mukainen malli. Malli sisälsi yleisimmät osat päiväkirjan kirjoittamisesta viiteaineiston luomiseen sekä ajojen parametrisuuteen. Ensimmäinen kehityskierros tarjosi hyvät kehitysmahdollisuudet muille kehityskierroksille, joiden aikana rakennettu ohjelmisto tosin päivittyi useita kertoja jatkuvan parantamisen kautta. Ensimmäiset varsinaiset testiajot toteutettiin kolmannen kehityskierroksen lopussa. Silloin saavutettiin varmuus, että viiteaineistot muodostuivat oikein. Niinpä varmistui, ettei monen päivän simulointia estyisi rikkinäisten aineistojen vuoksi.

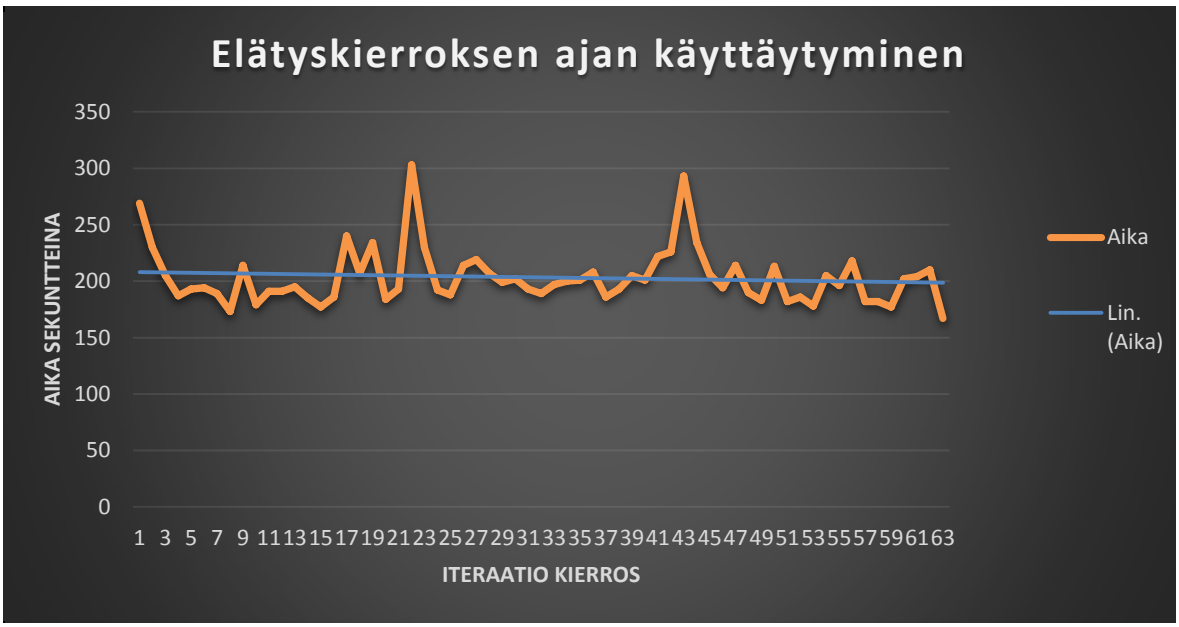
Tuotettua ohjelmistoratkaisua käytettiin testaamaan tietojärjestelmää. Toimeksiantajan edustaja tunsi tietojärjestelmän entuudestaan todella hyvin, mikä helpotti tuloksien analysointia sekä koko kehitystyöprosessia. Ajokierroksia suoritettiin muutamia kymmeniä kertoja, joiden aikana huomattiin tuotettuun ohjelmistoon liittyviä parannusideoita ja kehittämiskohteita. Testattavaan tietojärjestelmään saavutettiin todella nopeasti verifioinnit niille toiminnollisuuksille, joita viiteaineistojen avulla testattiin. Useiden kierrosten aikana huomattiin, että järjestelmä tämän hetkisessä versiossaan toimii sen määritettyjen toiminnollisuuksien mukaisesti.

Analysointi perustuu tallennettaviin loki- sekä aineistotietoihin, joiden avulla voidaan arvioida järjestelmän toimivuutta. Samalla laatutekijöistä tulee testattua siirrettävyyttä, tehokkuutta ja luotettavuutta. Käytettävyyden analysointi ei kuitenkaan ole sellainen, jolla automatisointia voisi tarkasti analysoida. Tosin käytettävyyttä voidaan miettiä niiden askelei-

den osalta, jonka aikana jokin toiminto suoritetaan. Tällöin käytettävyyteen liittyy osakseen ymmärrys käytettävästä järjestelmästä. Tämä näkyy automaatiossa siten, että ohjelmoidussa toteutuksessa suoritetaan ohjelmallisesti samat vaiheet kuin manuaalisessa toteutuksessa.

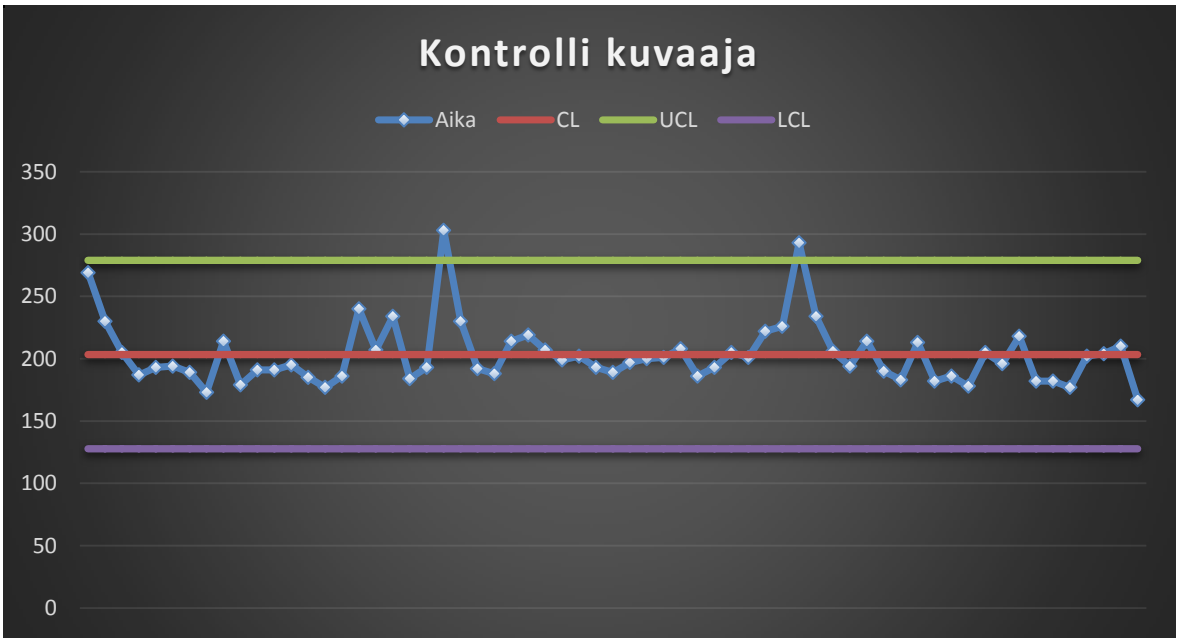
Jokaisella ajokierroksella tallennetaan kaikki järjestelmän tuottamat tiedot sekä siihen luodut viiteaineistot. Näitä aineistoja muodostuu eri tavalla riippuen siitä, miten järjestelmä toimii. Tässä tapauksessa muutamat aineistot voivat jäädä muodostumatta, mikäli viiteaineistojen esiehdot eivät täyty. Esimerkiksi maksussa ei voi olla sellaisia maksurivejä, joita ei ole muodostunut tietojärjestelmän kautta. Esiehtojen täyttymiseen liittyy myös satunnaistapahtumat, jotka voivat tapahtua ainoastaan silloin, kun testattavan järjestelmän kannalta se voisi tapahtua. Tietenkin aineiston voisi muodostaa joka kierroksella, mutta tällöin emme testaisi elinkaarta vaan toiminnollisuutta. Tarkoituksena on testata tietojärjestelmän elinkaarta päivien kuluessa eteenpäin automaation kautta ohjatusti.

Kuvasta 18 voidaan nähdä, että muutamaa piikkiä lukuun ottamatta, keskimäärin elätyskierros kestää vähän päälle 200 sekuntia nykyisessä tietojärjestelmäversiossa ja tuotetuilla viiteaineistoilla. Isoimmat piikit aiheutuvat kuukausiajosta, jossa testattava tietojärjestelmä tekee enemmän töitä taustalla. Myös muutamia erikoislaatuisia nousuja ja laskuja tapahtuu kierroksien 17-21, 49-53 ja 61-63. Näiden analysoimiseen tiedot löytyvät tarkemmin lokeista ja generoiduista aineistoista, joiden avulla saadaan testattavan ohjelmiston lisätietoja. Muuten käyttäytyminen tapahtuu suhteellisen rauhallisesti.



Kuva 18. Ajettujen elätyspäivien kesto sekunneissa

Aikaisemmasta taulukosta voidaan muodostaa myös prosessin kontrollimalli, jossa voidaan tarkemmin tutkia niitä pisteitä, jotka ovat hallitun muutoksen mukana. Tämä on esitetty kuvassa 19. Muutoksen yläraja ja alaraja on laskettu keskihajonnan mukaisesti ja kerrottu luvulla kolme. Tämän jälkeen ylärajan muodostamisessa on lisätty keskiarvoon summa ja alarajan laskemisessa keskiarvosta on vähennetty summa. Rajojen muodostamiseen tosin voidaan käyttää myös mielivaltaisia lukuja, mutta kuvan tapauksessa ne on toteutettu laskennan avulla. Kuvassa 19 esiintyy yhteensä kaksi erikoista pistettä, jotka eivät noudata muutoksen rajoja. Molemmat ovat selitettävissä järjestelmän sisäisillä syillä, jolloin prosessointia tapahtuu keskimääräistä enemmän järjestelmän sisällä. Näiden selvittäminen voidaan tutkia kerätystä lähdeaineistosta sekä käyttöliittymän kautta testattavasta tietojärjestelmästä. Ideana kuvassa on löytää kaikki ne tapaukset, jotka eivät noudata kontrollirajoja.



Kuva 19. Kontrollikuvaaja

Muuten ajot vahvistivat käsitystä järjestelmän käyttäytymisestä. Suunnitellut käyttäytymiset, jotka toimeksiantaja oli ennalta pohtinut, toteutuivat poikkeuksetta. Jo muutaman ajokierroksen datan jälkeen oli selvitetty muutama järjestelmässä oleva tapaus, jotka olivat herättäneet kiinnostusta.

Ennen kuin varsinainen kehitystyö opinnäytetyöprojektin osalta saatiin päätökseen, käytettiin jo luotua aineistojen generointimallia toisessa projektissa hyödyksi. Tämä lisäsi tuotetulle ohjelmistoratkaisulle yhden uuden ominaisuuden, joka ei sisältynyt alkuperäiseen opinnäytetyöprojektin suunnitelmaan. Uuden toiminnollisuuden toteutuessa tuotetun ohjelmistoratkaisun käyttötavat kasvoivat, ja ratkaisua pystyttiin käyttämään muissa projek-

teissa. Automaation kautta generoidut aineistot otettiin käyttöön toisen projektin tietojen alustusvaiheessa. Mikäli ohjelmistoratkaisua ei olisi ollut, tämä kaikki olisi pitänyt tuottaa käsin alustusvaiheessa.

8 Pohdinta, havainnot ja jatkokehittäminen

Opinnäytetyöprojektin ohjelmistokehitys koostui monen eri järjestelmän hallinnasta sekä ketterän kehittämisen metodista sekä ajattelutavoista. Ohjelmistoa, jonka kaikkia ominaisuuksia ei pystytä heti määrittelemään, mutta jonka kriittiset ominaisuudet tunnetaan ja pystytään määrittämään, tuntui käytännöllisempänä kehittää ohjelmisto ketterän ohjelmistokehityksen kautta. Tällöin on myös mahdollista löytää uusia entuudestaan tuntemattomia toimintoja, joita ei välttämättä heti tiedosteta sekä reagoida niiden tuottamaan mahdolliseen muutokseen.

Lean ja kanban tuottivat mielestäni lisäarvoa projektin etenemisen kannalta. Lisäämällä näitä ajatuksia kehitystyön luonne muuttui enemmän kehittäjäkriittisemmäksi, jolloin kehittäjänä pääsin tekemään päätöksiä toteutustavoista. Omassa työssä tämä ilmeni siten, että en enää pyrkinyt prototyyppeinä kovakoodattuihin rakenteisiin. Rakenteet muuttuivat enemmän dynaamisiksi ja parametrisoitaviksi, jolloin niiden testaaminen lähtökohtaisesi oli helpompaa eikä uudelleen tekemistä toteutettu usein. Toisaalta testivetoinen kehittäminen ei soveltunut itselleni kokemuksen puutteen johdosta. Testivetoisuus olisi Lean kehityksen mukaan parempi ratkaisu toteuttaa kehittämistyö turhan työn minimoimiseksi. Myös jatkuvien prototyyppien tuottaminen ennen varsinaista lähdekoodin lisäämistä muutti metodien rakennetta, kun kehittäjänä huomasin parempia ratkaisuja.

Projektin kannalta harmittavaa oli, että kehitystiimin muodosti käytännössä yksi henkilö. Niinpä aivan kaikkia ketterän kehittämisen monipuolisuuksia päivittäisessä työssä ei ollut mahdollista toteuttaa oppikirjan mukaisesti, kuten parikoodausta ja lähdekoodin katselmointia. Lisäksi lyhyitä päiväpalavereita ei pidetty scrumin mukaisesti vaan päiväpalaverit koostuivat lyhyistä 5-10 minuutin tapaamisista toimeksiantajan edustajan kanssa, joissa tosin käsiteltiin päiväpalaverin asiat: missä mennään, mitä tehtiin ja onko esteitä odotettavissa. Toisaalta ketterä kehittäminen sekä tuntui miellyttävämmältä ratkaisulta että tuloksena mielestäni saattoi olla parempi kuin vesiputousmallilla kehitetty tuote, varsinkin kun kaikkia tuoteominaisuuksia ei tunnettu eikä pystytty täysin määrittämään ennen työn aloittamista. Työstä tunnettiin kuitenkin keskeisemmät osa-alueet, ja ne pystyttiin hahmottamaan. Yksityiskohtainen määrittämien ei aina ollut tarpeen. Esimerkkinä voisi mainita tiedoston kirjoittamisen, jossa kirjoittaja pystyttiin luomaan tietämättä mitä se tulee kirjoittamaan, kun ainoastaan kirjoitettava teksti on parametrina. Niinpä yksityiskohtainen määrittäminen voitiin suorittaa myöhemmissä toteutusluokissa.

Inkrementaalisesti kehittäen ominaisuudet tuntuivat valmistuvan laadukkaammin, kun ajatus pysyi koko kehityksen ajan mukana. Myös jatkuva integrointi mahdollisti nopean pa-

lautteen saamisen, jolloin pystyttiin nopeasti reagoimaan, mikäli uusi toiminnollisuus rikkoi muita toimintoja. Analysoimalla lähdekoodia oma varmuus kehittäjänä myös kasvoi sekä varmistui siitä, että tuotettu lähdekoodi on laadukasta.

DevOps-tyyliinen kehittäminen liiketoiminnallisten vaatimusten ja intressien liittymiseksi kehitystyöhön auttoi erityisesti validointivaihetta. Kun oli selvää, millaisessa esitysmuodossa tiedon olisi oltava, oli helppoa käyttää tätä tietoa hyödykseen testien kirjoittamisessa validointiin. Viiteaineiston hahmottaminen koko liiketoimintaan avasi paremmin testattavan ohjelmiston käyttäytymistapaa sekä lisäsi tietoisuutta finanssialan toiminnasta. Lisäksi Bamboon tarjoama jatkuva integraatio edesauttoi testien ajamista muualla kuin omassa kehitysympäristössä sekä loppujen lopuksi automatisoi tuottamisketjun pitkälle. Myös jatkuva palautteen saaminen ohjasi omatoimisiin ratkaisuihin ja ratkaisumallien luomiseen, ja niiden testaaminen oli suhteettoman vaivatonta.

Kehittäjänä minulle tuli useita kertoja vastaan tilanteita kehitysvaiheissa, joita en ennen ollut kohdannut. Haasteina oli monenlaisia luovuutta vaativia ongelmia aina validoinnista tiedon järkevässä muodossa siirtämiseen eri prosessien välillä sekä automaation tuottaminen esimerkiksi kyselyjen tuloksien keräämisessä. Opin käyttämään paljon uusia tekniikoita ohjelmointikielestä, joista generiikka ja reflektio tulevat tulevaisuudessa olemaan itselleni kehittäjänä todella hyödyllisiä tekniikkoja. Samoin eri järjestelmien käyttäminen lisäsi tietojani versionhallinnan monipuolisuudesta ja ennen kaikkea jatkuvan integroinnin kautta opin enemmän ohjelmiston käyttäytymisestä eri tiloissa.

Kokonaisuuden hallinta oli huomattavasti helpompaa ja erityisesti mitattavampaa useamman järjestelmän kautta. Oppiminen uusista järjestelmistä toi myös kehittämiseen uusia näkökulmia, joita en välttämättä olisi itse huomannut. Esimerkiksi SonarQuben käyttäminen lähdekoodin laadun mittaamisessa avasi itselleni paremmin laadun käsitettä ohjelmiston sisäänrakennettuna ominaisuutena. Opin myös muutamia hyödyllisiä taitoja, joita en olisi välttämättä vielä muutamassa vuodessa itse oppinut ilman tämän järjestelmän tukea. Tilannetiedon sisäinen julkistaminen toimeksiantajan järjestelmissä helpotti työn seurantaa ja koko ajan tiedostettiin projektin tilanne.

Tuotettu ohjelmistoratkaisu saatiin opinnäyteprojektin kannalta valmiiksi. Jatkokehittämisen kohteita on useita, koska tietojärjestelmä mihin elinkaaritestaus toteutettiin sisältää useita ja monenlaisia toiminnollisuuksia ja lisäaineistoja. Esimerkkejä toiminnollisuuksista riittää uusien vakuutussopimusten generoidusta lisäämisestä sopimusten päättämiseen. Niitä järjestelmässä on useita eri tapoja toteuttaa. Opinnäytetyössä rakennettiin kuitenkin kaikki tarvittavat valmiudet, jotta jatkokehittäminen on mahdollista toteuttaa. Suurinta osaa

jo luoduista toiminnoista voidaan käyttää sellaisenaan, mutta käytännössä tämä kuitenkin tarkoittaa lisää ohjelmointia haastavampien toiminnollisuuksien testiaineiston muodostamiseen. Esimerkiksi erikoismääritettyjen viiteaineistojen automatisoituun generoimiseen joutuu väistämättä ohjelmoimaan uusia luokkia ja metodeja.

Suurimmat jatkokehittämistoimet tulisi kuitenkin suunnata uusien toiminnollisuuksien kehittämiseen. Uusilla toiminnollisuuksilla päästään suhteellisen nopeasti testaamaan uusia osia järjestelmästä, joita vielä ei ole automatisoidussa elinkaaritesteissä testattu. Opinnäytetyössä käytiin yleisimpiä tapauksia lävitse, jotta voidaan varmistua, että kehitetty uusi ohjelmisto kykenee suoriutumaan sille asetetuista tehtävistä. Samalla myös todennettiin, että automatisoiduilla ratkaisuilla voidaan hankkia säästöjä ja keskittää resursseja.

Toisaalta jatkokehittämisessä voisi myös toteuttaa ohjelmaan sisäisen analysointikomponentin. Komponentti voisi analysoida aineistojen luomiseen kulunutta aikaa, aineistojen määrää ja niihin käytettävää prosessoritehoja, muistin käyttöä, tietokantayhteyksiin kulunutta aikaa sekä suorittaa tietokantaan omia analysointikyselyjä. Komponentin avulla voitaisiin saada ennakkotietoja, joiden avulla voitaisiin tukea tehtyä analysointia.

Jatkokehittämisessä tulee kuitenkin muistaa se, että kaikkea ei ole välttämätöntä automatisoida. Automatisoinnin tulisi koskea niitä aineistoja ja työvaiheita, joissa automaatio on vaivatonta tuottaa sekä helposti ylläpidettävää. Automatisoinnilla on mahdollista saada sellaisia hyötyjä, joita manuaalisella aineiston tuottamisella on haastavaa saavuttaa. Tietyllä tavalla automatisoidut pitäisi olla käytettävissä myös muissa tietojärjestelmissä esimerkiksi parametrisoinnin avulla, jolloin tehdyt ratkaisut olisivat järjestelmäriippumattomia.

Mikäli kehitystiimin koko kasvaisi, voisi kehitystiimi aloittaa ensimmäisenä ohjelman lähdekoodin refaktoroinnin. Koska ohjelma on tällä hetkellä käytännössä toteutettu yhden kehittäjän näkökulmasta ja kokemuksesta, voisivat uudet näkökulmat ja lisäkokemus parantaa ohjelman suorituskykyä, dynaamisuutta sekä tuottaa lähdekoodiin enemmän laadukkuutta. Samalla löydettäisiin kaikki ne puutteelliset kommentit, jotka kaipaisivat lisäselvitystä. Tällä varmistettaisiin, että tulevatkin kehittäjät saisivat arvokasta tietoa ohjelman käyttäytymisestä eikä aikaa kuluisi lähdekoodin tutkimiseen niin paljon.

Eräs oikein käyttäjäystävällinen tapa olisi rakentaa käyttöliittymä, jossa töitä voisi käsin suunnitella. Nykyisessä toteutuksessa on turvauduttava tekstieditoriin ja omaan tietotaitoon. Käyttöliittymän avulla voitaisiin ratkaista töiden nopea generointi ja jopa näiden automatisointi robotin kautta. Tätä ei kuitenkaan työn tässä vaiheessa koettu tarpeelliseksi, mutta se tarjoaisi lisäarvoa ja käyttömukavuutta.

Käyttöliittymältä voisi myös lukea lähdeaineistoja ja lokeja. Nykyisessä versiossa nämä tallennetaan zip-tiedostoon, joka on tallennettu palvelimen asemalle. Käyttöliittymän kautta voitaisiin nopeammin käsitellä selaimen tiedostoja, ja mahdollisesti rakentaa myös arkistointiratkaisu, jossa jokaisen ajokierroksen tiedot tallennettaisiin omaan hakemistoon.

Elätyskomentosarjaa varten voitaisiin myös lisätä automatisoitu käyttöliittymärakenne, jossa käyttäjä voisi tuottaa käyttöliittymän kautta elätysskriptin. Hyötyinä tässä olisi käyttöliittymän avulla nähdä, mitä ohjelma tekisi ja väärässä hakemistossa suoriteltuilta komennoilta vältyttäisiin ehkä visuaalisuuden avulla. Nykyisessä versiossa komentosarja on kirjoitettu tekstieditorilla ja vaatii käyttäjältä kykyä hahmottaa komentosarjojen kulku vaihe vaiheelta.

Opinnäytetyö opetti minulle paljon automatisoinnin eduista ja vaatimuksista sekä myös itsenäisestä työskentelystä. Mielestäni ohjelmistolle asetetut tavoitteet saavutettiin opinnäytetyöprojektin aikana hyvin ja keskeisimmät jatkokehittämiskohteet on tunnistettu. Kehitetyt automatisoidut ratkaisut helpottavat hyväksymistestausvaihetta ja vähentävät resurssien kuormittamista. Myös käytettävissä olevaa tietojärjestelmää tunnetaan paremmin. Opinnäytetyön aikana rakennetut puitteet mahdollistavat ohjelmiston käyttämisen myös muissa tietojärjestelmissä pitkälle viedyn parametrisuutensa johdosta.

Lähteet

Agile Allicance. 2015. Iteration. Luettavissa: https://agilealliance.org/glossary/iteration/. Luettu: 25.3.2016

Allan, K. 2008. Changing Software Development. John Wiley & Sons. Ltd. England.
Apache Maven. 2016. Welcome to Apache Maven. Luettavissa: https://maven.apache.org/. Luettu:25.3.2016.

Beck, K., Beedle, Mike., Bennekum, A V., Cockburn, A., unningham, W C., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, Jon., Marick, B., Martin, R C., Mellor, S., Schwaber, K., Sutherland, J. & Thomas, Dave. Ketterän ohjelmistokehityksen julistus. 2001. Luettavissa: http://agilemanifesto.org/iso/fi/. Luettu: 26.3.2016

Black, R., Graham, D., Evans, I. & Veenendaal, E V. 2009 Foundation of Software Testing ISTQB certification. Zrinski dd. Croatia.

Blum, R. 2008. Linux® Command Line and Shell Scripting Bible. Wiley Publishing Inc. United States of America.

Charalampos, A S. 2012. Sonar Code Quality Testing Essentials. Packt Publishing Ltd. United Kingdom.

Cooke, J L. 2012. Everything you want to know about Agile. IT Governance Publishing. United Kingdom.

Fowler, M. 2013. Continuous Delivery. Luettavissa: http://martinfowler.com/bliki/ContinuousDelivery.html. Luettu: 26.03.2016

Gao, J , Tsai, H-S & Wu, Y. 2003. Testing and Quality Assurance for Component-Based Software. Artech House Books. London

Git. 2016. Getting Started – About Version Control. Luettavissa: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control. Luettu: 27.3.2016.

GoodPasture, J C. 2014. Project Management the Agile Way. J.Ross publishing. United States of America.

Graham, D & Fewster, M. 1999. Software Test Automation effective use of test execution tools. Luettavissa: http://read.pudn.com/downloads11/ebook/44105/Software%20Test%20Automation.pdf. Luettu: 20.04.2016

Harju, J. & Juslin, J. 2009. Java ohjelmointi. Gummeruksen kirjapaino Oy. Jyväskylä
Hibbs, C., Jewett, S. & Sullivan, M. 2009. The Art of Lean Software Development. O'Reilly. United States of America.

Hobbs, D P. 2014. Lean Manufacturing Implemantion. J. Ross Publishing Inc. United States of America.

Holzner, S. 2001. Java 2 Black Book. The Coriolis Group. United States Of America.

ISO 9126-1. 2000. Information technology — Software product quality — Part 1: Quality model. Final Draft. Luettavissa: http://www.cse.unsw.edu.au/~cs3710/PMmaterials/Resources/9126-1%20Standard.pdf. Luettu: 25.03.2016.

ISO/IEC 25010. 2011. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models Luettavissa: https://www.iso.org/obp/ui/#iso:std:35733:en. Luettu: 25.3.2016

Isomäki, M., Jokela, T., Kaisti, M., Känsälä, M., Könnölä, K., Lehtonen, T., Mäkilä, T., Rantala, V., Suomi, S., Tuomivaara, S., & Ylitolva, M. 2014. Sulautettujen järjestelmien ketterä käsikirja. Painosalama Oy. Helsinki. Luettavissa: http://www.doria.fi/bitstream/handle/10024/99142/Sulautettujen_jarjestelmien_kettera_kasikirja_Painos1.pdf?sequence=2. Luettu: 13.3.2016

Juran, J. M., Godgrey, A. B., Hoogtoel, R. E & Schilling, E. G. 1999. Juran's Quality Handbook. 5th Edition. McGraw-Hill. Unites States of America

Kainulainen, P. 2013. Unit Testing of Spring MVC Controllers: Configuration. Luettavissa: http://www.petrikainulainen.net/programming/spring-framework/unit-testing-of-spring-mvc-controllers-configuration/. Luettu: 27.3.2016.

Koch, A. 2004. Agile Software Development. Artech House Books. London.

Lakshman, S. 2011. Linux Shell Scripting Cookbook. Packt Publishing Ltd. United Kingdom.
Lalou, J. 2013. Apache Maven Dependency Management. Packt Publishing Ltd. United Kingdom.

Leffingwell, D. 2008. Scaling Software Agility: Best Practices for Large Enterprises. Addison-Wesley. India.

Leino, J & Thorström, T. 2015. DevOps. Luettavissa: https://solinor.fi/devops/. Luettu: 2.4.2016

Lillrank, P., Lemetti, P., Järvinen, P., Malmi, T. & Virtanen, T. 2003. Laatukustannuslaskenta: käyttötarkoitus ja menetelmät käytännön työkirja yrityskäyttöön ja opiskeluun. 3 painos. Monikko Oy. Espoo.

Lipponen, T. 1993. Laatujohtaminen laatujohtamistyökalujen valinta ja soveltaminen. Gummeruksen kirjapaino Oy. Jyväskylä.

Mansio L. 2015. Testauksen automatisointi on haasteellista mutta palkitsevaa. Luettavissa: http://www.pcuf.fi/sytyke/lehti/kirj/st20051/ST051-08A.pdf. Luettu: 31.3.2016.

Poimala, S & Tolvanen, P. 2013. Ketteryys haltuun: Ketterän kehityksen yleiset periaatteet Luettavissa: http://www.meteoriitti.com/2013/06/06/ketteryys-haltuun-ketteran-kehityksen-yleiset-periaatteet/. Luettu: 13.3.2016

Poppendieck, M & Poppendieck, T. 2007. Implementing Lean Software Development: From Concept to Cash. Addison-Wesley. United States of America.

Rother, M. & Shook, J. 2003. Learning to See Value-Stream Mapping to Create Value and Elimante Muda. The Lean Enterprise Insitute Inc. United States of America.

Saddington, P. 2012. The Agile Pocket Guide. John Wiley & Sons, Inc. United States Of America.

Schwaber, K & Sutherland, J. The Scrum Guide. 2013. Luettavissa: http://www.scrumguides.org/docs/scrumguide/v1/scrum-guide-us.pdf. Luettu: 13.3.2016

Scrum Alliance. 2014 The Scrum Guide. Luettavissa: https://www.scrumalliance.org/why-scrum/scrum-guide. Luettu: 3.4.2016

Somasundaram, R. 2013. Git: Version Control for Everyone. Packt Publishing Ltd. United Kingdom.

Spring-Boot. 2016. Spring Boot. Luettavissa: http://projects.spring.io/spring-boot/. Luettu: 25.03.2016.

Spring Boot Guide. 2016a. 13 Build systems. Part III Using Spring Boot. Luettavissa: https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-build-systems.html. Luettu: 26.3.2016.

Spring Boot Guide. 2016b. 40 Testing Part IV Spring Boot Features Luettavissa: https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-testing.html. Luettu: 27.3.2016

Spring.Io. 2016. Building an Application with Spring Boot. Luettavissa: https://spring.io/guides/gs/spring-boot/. Luettu: 27.3.2016.

Srirangan, I. 2011. Apache Maven 3 Cookbook. Packt Publishing Ltd. United Kingdom.

Swartout, P. 2012. Continous Delivery and DevOps: A Quickstart guide. Pack Publishing Ltd. United Kingdom.

The W.Edwards Deming Institute. 2016. The PDSA Cycle. Luettavissa: https://www.deming.org/theman/theories/pdsacycle. Luettu: 10.4.2016

Vesterholm, M. & Kyppö, J. 2003. Java Ohjelmointi. Talentum. Helsinki
Vuori, M. 2010. Leanistä ja testauksesta. Luettavissa: http://www.mattivuori.net/julkaisuluettelo/liitteet/leanista_ja_testauksesta.pdf. Luettu: 13.3.2010

WSOY. 2006. Facta tietosanakirja. Werner Södeström Osakeyhtiö. Porvoo.

Waters, K. 2010. 7 Key Principles of Lean Software Development. Luettavissa: http://www.allaboutagile.com/lean-principles-3-create-knowledge/. Luettu: 13.3.2016

Williams, N. S. 2014. Professional Java for Web Applications. Wrox. United States of America.

Liitteet

Liite 1. Ketterän kehittämisen 12 pääperiaatetta

	English	Suomeksi
1	**Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.**	Tärkeintä on tyydyttää asiakas julkaisemalla aikaisin ja jatkuvasti laadukkaalla ohjelmistolla.
2	**Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.**	Hyväksytään muuttuvat vaatimukset, jopa kehitystyön loppuvaiheessa, ketterä prosessi valjastaa muutokset asiakkaan kilpailueduksi.
3	**Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.**	Toimitetaan toimivia ohjelmistoja säännöllisesti, mielellään lyhyin aikajaksoin (muutamasta viikosta kuukauteen)
4	**Business people and developers must work together daily throughout the project.**	Liiketoiminnan ammattilaisten ja ohjelmiston kehittäjien täytyy työskennellä yhdessä päivittäin koko projektin ajan
5	**Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.**	Rakennetaan projektit motivoituneiden yksilöiden ympärille ja annetaan heille ympäristö ja tuki, sekä luotetaan että he saavat työn tehtyä
6	**The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.**	Tehokkain tapa välittää tietoa kehitystiimille ja kehitystiimin sisällä on kasvokkain tapahtuva keskustelu
7	**Working software is the primary measure of progress.**	Toimiva ohjelmisto on ensisijainen edistymisen mittari
8	**Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.**	Ketterät menetelmät suosivat kestävää kehitystä. Rahoittajien, kehittäjien ja käyttäjien tulisi pystyä pitämään toistaiseksi yllä tasainen työtahti.
9	**Continuous attention to technical excellence and good design enhances agility.**	Jatkuva huomion kiinnittäminen tekniseen laatuun, sekä hyvää rakenteeseen, lisää ketteryyttä
10	**Simplicity--the art of maximizing the amount of work not done--is essential.**	Yksinkertaisuus – taito maksimoida työn määrä, jota ei tarvitse tehdä, on olennaista
11	**The best architectures, requirements, and designs emerge from self-organizing teams.**	Parhaat arkkitehtuurit, vaatimukset ja rakenteet tulevat itseorganisoituvista tiimeistä
12	**At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.**	Säännöllisin väliajoin tiimi miettii miten voisi tulla entistä tuottavammaksi, ja muokkaa toimintaansa sen mukaisesti

Beck ym. Principles behind the Agile Manifesto. Luettavissa: http://agilemanifesto.org/principles.html. Luettu: 26.3.2016

Liite 2. Periytyminen, rajapinnan toteuttaminen ja reflektio

```java
package fi.sphv.example.interfaces;
//Esitetään rajapinta luokka
public interface Book {
    public Book getBook();  //Metodi, joka palauttaa rajapinta luokan ilmentymän
}
```

Lähdekoodi 1. Rajapintaluokka

```java
package fi.sphv.example.bean;
//Abstakti luokka
public abstract class AbstrackBook {
    //Abstaktri metodi, jossa tarkoituksena tulostaa kirjan nimi
    public abstract void printBookName();
    //Attribuutteja
    private String nimi;
    private String isbn;

    //Get ja set metodit, konstruktorit....

}
```

Lähdekoodi 2. Abstrakti luokka

```java
package fi.sphv.example.bean;

import fi.sphv.example.interfaces.Book;
//Luokka toteuttaa Book rajapinnan ja perii AbstrackBook luokan
public class ScienceBook extends AbstrackBook implements Book{
    //Rajapinta luokan metodi
    public Book getBook() {
        return this;
    }
    //Abstraktin luokan metodi
    @Override
    public void printBookName() {
        System.out.println(this.getNimi());
    }
    //Konstruktori
    public ScienceBook(String name, String isbn) {
        super(name, isbn);
    }
    //Konstruktori
    public ScienceBook() {
        super();
    }
    //ToString metodi, jossa tulostetaan luokan tiedot
    @Override
    public String toString() {
        return "ScienceBook [" + getNimi() + ", " + getIsbn() + "]";
    }
}
```

Lähdekoodi 3. Aliluokka toteuttaa rajapinnan sekä periytyy yliluokasta

```java
package fi.sphv.example.bean;

import fi.sphv.example.interfaces.Book;
//Luokka toteuttaa Book rajapinnan ja perii AbstrackBook luokan
public class ITBook extends AbstrackBook implements Book{
    //Rajapinta luokan metodi
    public Book getBook() {
        return this;
    }
    //Abstraktin luokan metodi
    @Override
    public void printBookName() {
        System.out.println(this.getNimi());
    }
    //Konstruktori
    public ITBook(String name, String isbn) {
        super(name, isbn);
    }
    //Konstruktori
    public ITBook() {
        super();
    }
    //ToString metodi, jossa tulostetaan luokan tiedot
    @Override
    public String toString() {
        return "ScienceBook [" + getNimi() + ", " + getIsbn() + "]";
    }
}
```

Lähdekoodi 4. Aliluokka toteuttaa rajapinnan sekä periytyy yliluokasta

```java
package fi.sphv.example.application;

import fi.sphv.example.bean.AbstrackBook;
//Luokka, jossa demostroidaan periytymistä ja reflektiota
public class App {
    public static void main(String[] args) {
        //Periytyminen
        AbstrackBook book = new ScienceBook("Planeetat", "978-951-98548-9-2");
        book.printBookName();//Konsoliin "Planeetat"
        //Mikäli book on ScienceBook ilmentymä
        if (book instanceof ScienceBook) {
            ScienceBook scienceBook = (ScienceBook) book;
            ScienceBook scienceCopy = (ScienceBook) scienceBook.getBook(); //Kopiodaan tiedot
            if(scienceBook.equals(scienceCopy)){    //Jos on scienceBook on yhtä kuin scienceCopy
                System.out.println(true);   //Tosi
            } else {
                System.out.println(false);  //Epätosi
            }
        }
        //Reflektion käyttö
        try{
        ITBook itbook = new ITBook();
        Class<?> clazz = Class.forName(itbook.getClass().getCanonicalName());
        Book reflectionBook = (Book) clazz.newInstance();   //Book rajapinnan toteuttavan luokan uusi ilmentymä
        if (reflectionBook instanceof ITBook){  //Jos reflectionBook on ITBook ilmentymä
            System.out.println(true);  //Tosi
        }else{
            System.out.println(false); //Epätosi
        }
        } catch(ClassNotFoundException e){
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
}
```

Lähdekoodi 5. Ajoluokka, jossa ensin luodaan aliluokan ilmentymä yliluokan kautta. Jälkimmäisessä ladataan luokka, joka toteuttaa rajapinnan

Liite 3. Parametrisoitava luokka

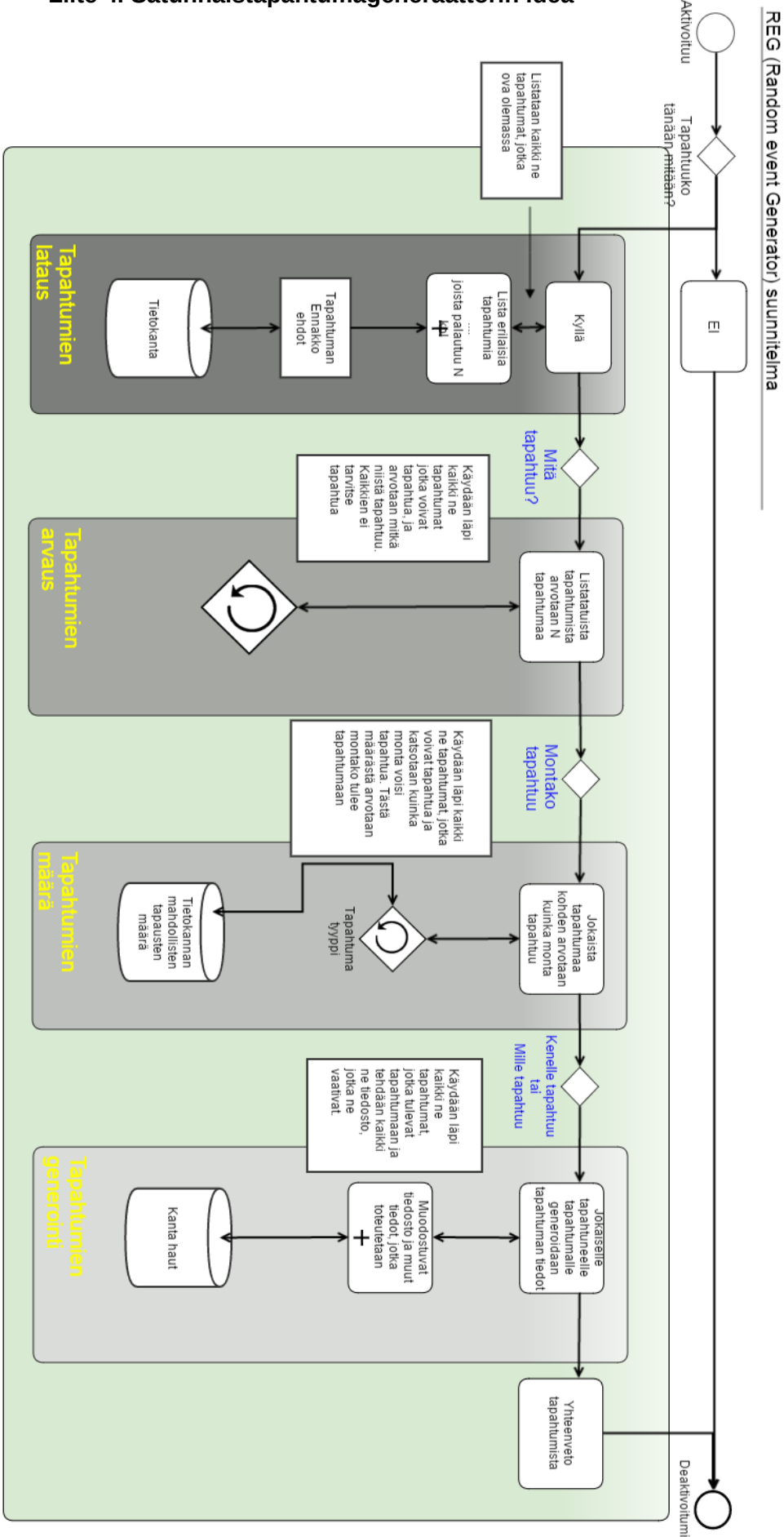
```
package fi.sphv.example.application;
//Luodaan luokka, jossa T toimii tyyppi parametrinä
public class Container<T> {
    //T tyyppinen parametri
    private T t
    //Hakee Container ilmentymän T
    public T getT(){
        return t;
    }
    //Asetetaan T tyyppinen ilmentymä Container olioon
    public void setT(T t){
        this.t = t;
    }
    //Staattinen main metodi
    public static void main(String[] args){
        Container<String> stringContainer = new Container<String>();//Alustetaan Container tyyppinä String
        String str = "Hei maailma";                                  //Alustetaan String arvolla "Hei maailma"
        stringContainer.setT(str);                                   //Asetetaan Containeriin str olio
        if(stringContainer.getT().equals(str)){                      //Jos Container olion teksti on sama kuin str
            System.out.printLn(true);
        }else{
            System.out.printLn(false);
        }
    }

}
```

Parametrisoitavassa luokassa parametrityyppi esitetään muuttujana *T*. *T* voi tyyppinä primäärinen tietotyyppi tai objektista periytyvä tietotyyppi. Myös itse luomat tietotyypit käyvät parametriksi, mikäli ne ovat periytyneet objekti luokasta.

Parametrisyys luokassa mahdollistaa sen, että jokaiselle luokalle ei tarvitse kirjoittaa samankaltaisia metodeja useasti. Esimerkki tällaisesta toiminnollisuudesta on esimerkiksi objektien hakeminen listasta. Toinen esimerkki voisi olla objektien kirjoittaminen esimerkiksi XML-muotoon, jossa objekti muodostaa omista attribuuteistaan ja niiden arvoista XML-elementin.

Liite 4. Satunnaistapahtumageneraattorin idea



Liite 5. Elätysmoottori komentosarjan toimintamallinnus